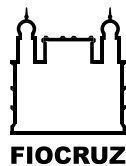


Programação Paralela de Aceleradores em OpenCL

Douglas A. Augusto

daa@fiocruz.br



ERAD-RJ – LNCC

Agosto/2015

Conteúdo

Conteúdo

1. Introdução & contextualização

Conteúdo

1. Introdução & contextualização
2. Arquitetura de dispositivos

Conteúdo

1. Introdução & contextualização
2. Arquitetura de dispositivos
3. A linguagem OpenCL

Justificativa

Por que GPU?

Justificativa

Por que GPU?

- Altíssimo poder computacional
- Excelente razão:
 - $\text{c\^omputo}/\text{energia}$
 - $\text{c\^omputo}/\text{pre\^co}$
 - $\text{c\^omputo}/\text{volume}$

Justificativa

Por que GPU de um único fabricante?

Justificativa

Por que GPU de um único fabricante?

- Existem vários fabricantes de GPUs *programáveis*:

AMD (Radeon R9 Fury X ~ 8600 GFlop/s)

ARM (Mali T880 ~ 600 GFlop/s)

Imagination (PowerVR GT7900 ~ 820 GFlop/s)

Intel (Pro Graphics 580 ~ 1325 GFlop/s)

NVIDIA (Titan X ~ 6140 GFlop/s)

Qualcomm (Adreno 430 ~ 390 GFlop/s)

⋮

Choose a test:

Face detection



1 to 20 of 2169



1

2

3

4

5

6

7

8

9

10



Show 20



	 NVIDIA GeForce GTX 980 Ti	199.686 mPixels/s
	OpenCL 2015.06.09 NVIDIA GeForce GTX 980 Ti	
	 NVIDIA GeForce GTX TITAN X	196.929 mPixels/s
	OpenCL 2015.07.17 NVIDIA GeForce GTX TITAN X	
	 NVIDIA Quadro M6000	192.248 mPixels/s
	OpenCL 2015.07.14 NVIDIA Quadro M6000	
	 NVIDIA GeForce GTX 980	154.395 mPixels/s
	OpenCL 2014.09.15 NVIDIA GeForce GTX 980	
	 AMD Radeon R9 Fury X	155.307 mPixels/s
	OpenCL 2015.06.27 AMD Radeon R9 Fury X	

Choose a test: Face detection

1 to 20 of 2169

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Show 20

		NVIDIA GeForce GTX 980 Ti	199.686 mPixels/s
	2015.06.09	NVIDIA GeForce GTX 980 Ti	
		NVIDIA GeForce GTX TITAN X	196.929 mPixels/s
	2015.07.17	NVIDIA GeForce GTX TITAN X	
		NVIDIA Quadro M6000	192.248 mPixels/s
	2015.07.14	NVIDIA Quadro M6000	
		NVIDIA GeForce GTX 980	154.395 mPixels/s
	2014.09.15	NVIDIA GeForce GTX 980	
		AMD Radeon R9 Fury X	155.307 mPixels/s
	2015.06.27	AMD Radeon R9 Fury X	

Choose a test: Bitcoin Mining

1 to 20 of 1304

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Show 20

		AMD Radeon™ R9 Series	919.406 mHash/s
	2015.06.19	AMD Radeon™ R9 Series	
		AMD Fijl	889.242 mHash/s
	2015.07.05	AMD Fijl	
		AMD Radeon R9 Kung Fury X	866.796 mHash/s
	2015.06.27	AMD Radeon R9 Kung Fury X	
		AMD Radeon R9 Fury X	857.575 mHash/s
	2015.06.27	AMD Radeon R9 Fury X	
		AMD Radeon™ R9 Fury Series	849.126 mHash/s
	2015.07.14	AMD Radeon™ R9 Fury Series	
		NVIDIA GeForce GTX 980 Ti	782.097 mHash/s
	2015.06.09	NVIDIA GeForce GTX 980 Ti	

Justificativa

Por que apenas GPU?

Justificativa

Por que apenas GPU?

- Por que não usar todos os dispositivos computacionais?
- Poderiam ser usados simultaneamente, agregando poder computacional
- A eficiência de uma arquitetura depende do padrão de processamento

Justificativa

Por que apenas GPU?

- Por que não usar todos os dispositivos computacionais?
- Poderiam ser usados simultaneamente, agregando poder computacional
- A eficiência de uma arquitetura depende do padrão de processamento

Tendência:

- CPUs convencionais + aceleradores (sistema heterogêneo)

The Green500 List

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,389.82	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	34.58
2	3,631.70	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,459.46	SURFsara	Cartesius Accelerator Island - Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4x FDR, Nvidia K40m	44.40
5	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66

Junho/2014

The Green500 List

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,389.82	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	34.58
2	3,631.70	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,459.46	SURFsara	Cartesius Accelerator Island - Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4x FDR, Nvidia K40m	44.40
5	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66

Novembro/2014

The Green500 List

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	5,271.81	GSI Helmholtz Center	L-CSC - ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150 Level 1 measurement data available	57.15
2	4,945.63	High Energy Accelerator Research Organization /KEK	Suiren - ExaScaler 32U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC	37.83
3	4,447.58	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	35.39
4	3,962.73	Cray Inc.	Storm1 - Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, Nvidia K40m Level 3 measurement data available	44.54
5	3,631.70	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62

Junho/2015

The Green500 List

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	7,031.58	RIKEN	Shoubu - ExaScaler-1.4 80Brick, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SC	50.32
2	6,842.31	High Energy Accelerator Research Organization /KEK	Suiren Blue - ExaScaler-1.4 16Brick, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband, PEZY-SC	28.25
3	6,217.04	High Energy Accelerator Research Organization /KEK	Suiren - ExaScaler 32U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC	32.59
4	5,271.81	GSI Helmholtz Center	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150	57.15
5	4,257.88	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	39.83

Processadores emergentes

- PEZY-SC (Peta Exa Zetta Yotta)
 - ◆ 1024 núcleos, ~ 3000 GFlop/s (SP)



Processadores emergentes

- PEZY-SC (Peta Exa Zetta Yotta)
 - ◆ 1024 núcleos, ~ 3000 GFlop/s (SP)



- PEZY-SC2
 - ◆ 4096 núcleos, ~ 16400 GFlop/s (SP)

Processadores emergentes

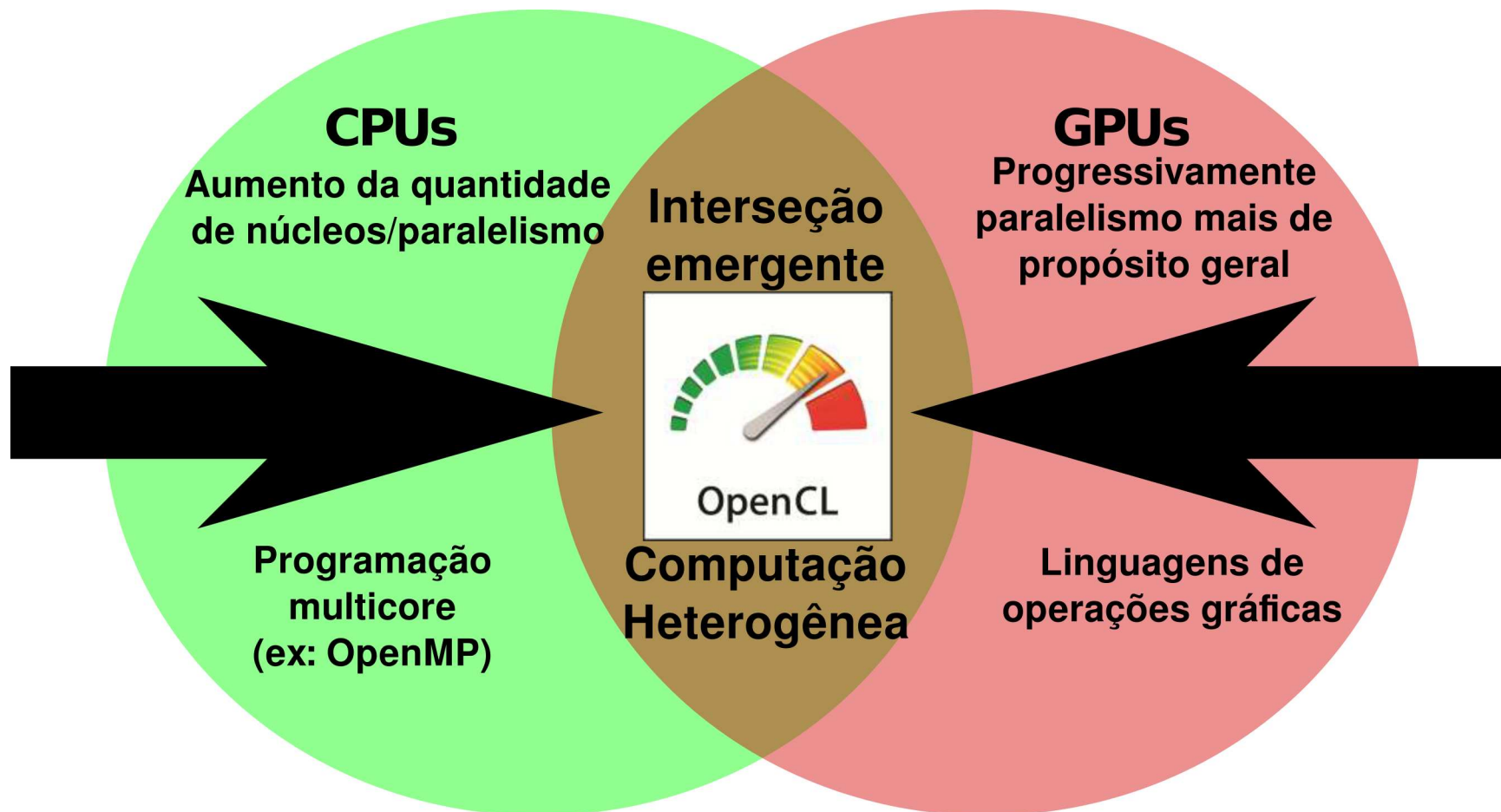
- PEZY-SC (Peta Exa Zetta Yotta)
 - ◆ 1024 núcleos, ~ 3000 GFlop/s (SP)



- PEZY-SC2
 - ◆ 4096 núcleos, ~ 16400 GFlop/s (SP)
- Altera Stratix-10 (FPGA)
 - ◆ ~ 9300 GFlop/s (SP)

OpenCL | Open Computing Language

“Padrão aberto para a programação paralela de sistemas heterogêneos”



OpenCL | Open Computing Language

Características:

- Provê interface *homogênea* para a exploração da computação paralela *heterogênea*
 - abstração do hardware
 - CPUs (AMD, ARM, IBM, Intel), GPUs (AMD, Nvidia, Intel, ARM, Imagination), APU, MIC, FPGAs, Epiphany, DSPs
- Padrão *aberto*
 - especificação mantida por vários membros
 - gerenciada pelo grupo *Khronos*

OpenCL | Open Computing Language

Características (cont.):

- Alto desempenho
 - possui diretivas de baixo nível para uso eficiente dos dispositivos
 - alto grau de flexibilidade
- Multi-plataforma
 - disponível em várias classes de hardware e sistemas operacionais
- Código *portável* entre *arquiteturas e gerações*

OpenCL | Open Computing Language

Características (cont.):

- Paralelismo de *dados* (“SIMD”) e *tarefa* (“MIMD”)
- Especificação baseada nas linguagens C e C++
- Define garantias para operações em ponto flutuante:
 - resultados consistentes independentemente do dispositivo
- Integração com outras tecnologias (ex: OpenGL)

WebCL | Paralelismo na Web

“OpenCL acessível via navegador”

WebCL | Paralelismo na Web

“OpenCL acessível via navegador”

- O objetivo é permitir que aplicações Web explorem todos os recursos computacionais disponíveis em um sistema heterogêneo
- É basicamente uma interface (em JavaScript) que acessa o OpenCL
- Pode ser integrado com o WebGL

História

- ~**2003**: GPUs começam a adquirir características de propósito geral: a era da programabilidade

História

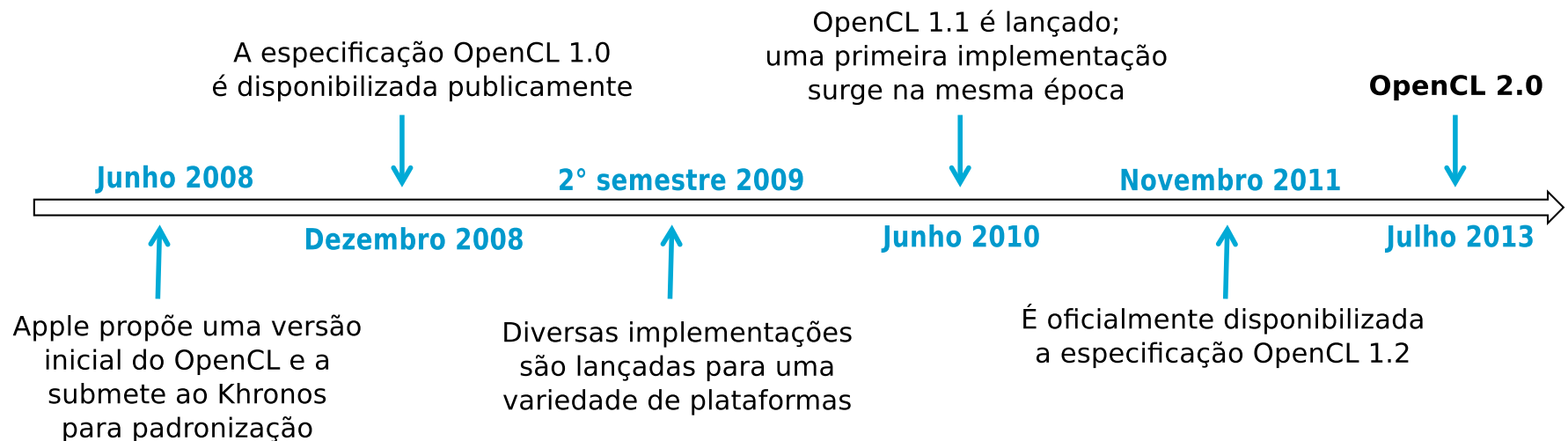
- ~**2003**: GPUs começam a adquirir características de propósito geral: a era da programabilidade
- **2003–2008**: Cenário GP-GPU fragmentado, com várias soluções proprietárias e míopes

História

- **~2003:** GPUs começam a adquirir características de propósito geral: a era da programabilidade
- **2003–2008:** Cenário GP-GPU fragmentado, com várias soluções proprietárias e míopes
- **2008:** Apple enxerga a oportunidade, intervém e desenvolve uma interface padronizada para computação GP-GPU em diferentes plataformas de hardware

História

- **~2003:** GPUs começam a adquirir características de propósito geral: a era da programabilidade
- **2003–2008:** Cenário GP-GPU fragmentado, com várias soluções proprietárias e míopes
- **2008:** Apple enxerga a oportunidade, intervém e desenvolve uma interface padronizada para computação GP-GPU em diferentes plataformas de hardware



História



Suporte da indústria em 2008

História



© Copyright Khronos Group, 2010 - Page 2

Suporte da indústria em 2010

OpenCL 1.2, 2.0 e 2.1

OpenCL 1.2:

- Particionamento de dispositivo (*Device partitioning*)
- Representação intermediária

SPIR — *Standard Portable Intermediate Representation*

OpenCL 1.2, 2.0 e 2.1

OpenCL 1.2:

- Particionamento de dispositivo (*Device partitioning*)
- Representação intermediária

SPIR — *Standard Portable Intermediate Representation*

OpenCL 2.0:

- Memória virtual compartilhada (*Shared Virtual Memory*)
- Paralelismo dinâmico (*Dynamic Parallelism*)
- Objetos *pipe*
- Diretivas atômicas (*C11 Atomics*)

OpenCL 1.2, 2.0 e 2.1

OpenCL 2.1 (Março/2015):

- Suporte ao C++14 (subconjunto)
 - ◆ Funções lambda, classes, templates, sobrecarga de operadores, etc.
- SPIR-V
 - ◆ Incorporado ao cerne da especificação
 - ◆ Unificação de operações computacionais (OpenCL) e gráficas (Vulkan)

Exemplos de Aplicações

Visualização/edição/manipulação de vídeo:

- VLC, FFmpeg, Adobe Premiere, Apple Final Cut Pro X, Sony Vegas Pro, Cyberlink PowerDirector, Corel VideoStudio Pro X4

Imagem:

- Adobe PhotoShop, GIMP, ImageMagick

Modelagem/Renderização 3D:

- Blender

Escritório:

- LibreOffice

Exemplos de Aplicações (II)

Computação científica:

- Matlab, Mathematica, ViennaCL, Folding@home, cIS-parse

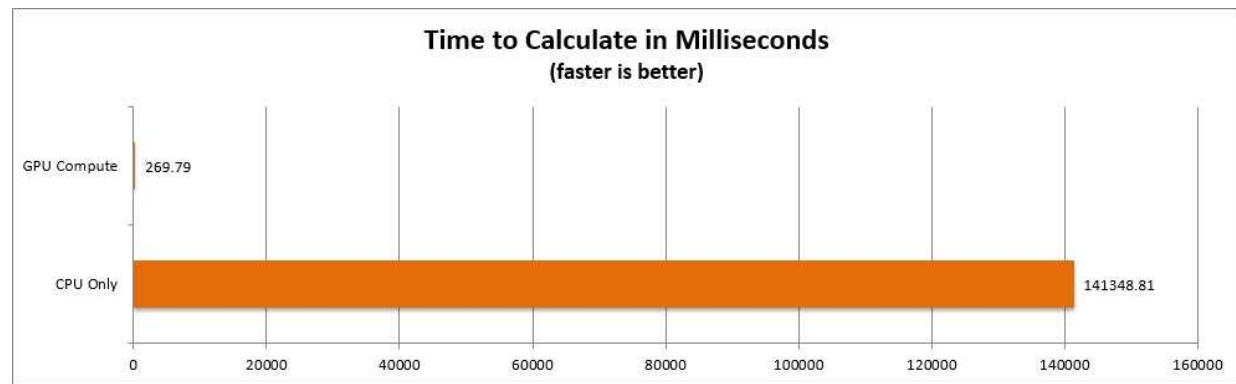
Ferramentas/utilitários:

- Winzip, Pyrit, oclHashCat, cRARk, etc.

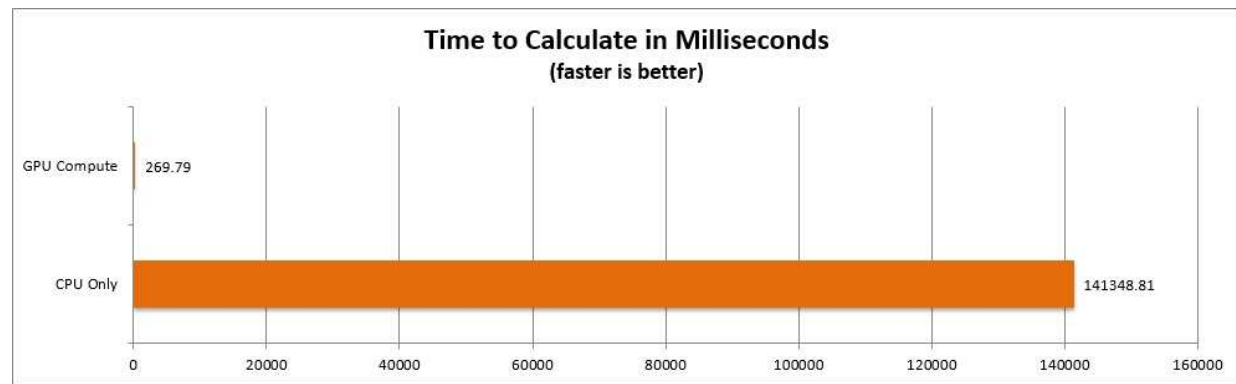
Jogo de computador:

- Battlefield

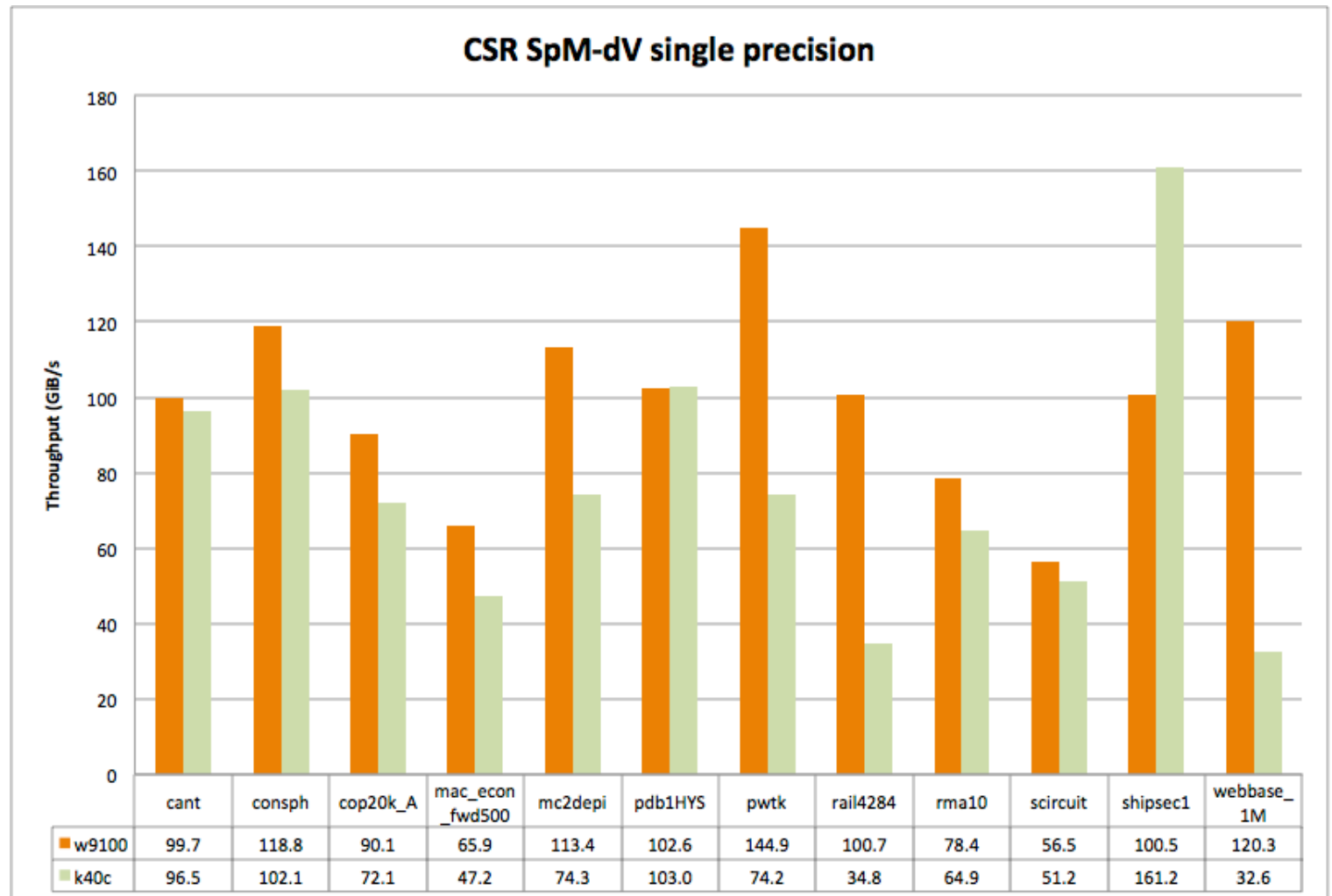
LibreOffice



LibreOffice



clSparse



Contextualização

OpenCL × CUDA

São tecnologias com alta interseção:

- Propósito parecidos
 - OpenCL foi influenciado por CUDA: ponto inicial
- Nível comparável de complexidade:
 - funcionalidades no que tange às GPUs
 - nível da linguagem
 - custo de engenharia de software
- Comparativamente mesmo desempenho

OpenCL × CUDA

Porém o CUDA:

- É uma tecnologia proprietária da Nvidia
- Não visa a computação heterogênea
- Desenvolvida especificamente para as GPUs Nvidia

OpenCL × OpenMP

OpenMP:

- Paralelismo tradicionalmente focado em CPU
- Mais alto nível:
 - programação mais simples, porém limitada/menos flexível
 - ganho de desempenho usualmente sub-ótimo

OpenCL × MPI

São tecnologias ortogonais:

- OpenCL: paralelismo *local*
 - usualmente memória compartilhada
- MPI: paralelismo *distribuído*
 - memória distribuída
- Podem ser combinadas: paralelismo em dois níveis

OpenCL × MPI

São tecnologias ortogonais:

- OpenCL: paralelismo *local*
 - usualmente memória compartilhada
- MPI: paralelismo *distribuído*
 - memória distribuída
- Podem ser combinadas: paralelismo em dois níveis
- OpenCL + MPI transparentemente?
 - *Virtual OpenCL, dOpenCL, Hybrid OpenCL, SnuCL*

Possível cenário futuro

Convergência para as abordagens:

- **MPI**
 - ◆ paralelismo distribuído

Possível cenário futuro

Convergência para as abordagens:

- **MPI**

- ◆ paralelismo distribuído

- **OpenMP/OpenACC**

- ◆ paralelismo incremental/fácil

Possível cenário futuro

Convergência para as abordagens:

- **MPI**

- ◆ paralelismo distribuído

- **OpenMP/OpenACC**

- ◆ paralelismo incremental/fácil

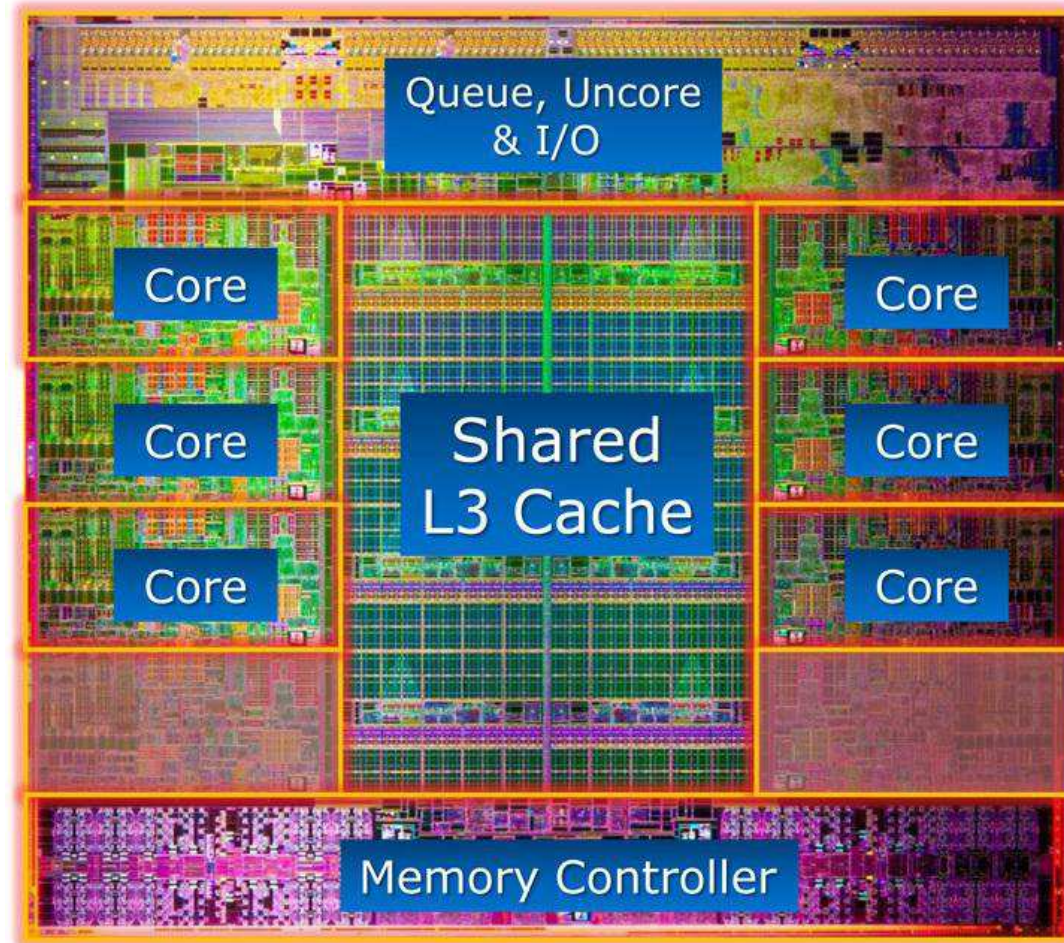
- **OpenCL**

- ◆ paralelismo massivo heterogêneo
- ◆ base para dezenas de outras linguagens/notações de mais alto nível

Arquiteturas CPU & Aceleradores

Arquitetura CPU

Intel Core i7-3690X: 160 GFLOP/s (DP)



Processo	Dimensão	Transístores	Núcleos	Frequência	Consumo
32nm	435mm ²	2,27 Bilhões	6 físicos e 6 lógicos	3,33GHz	130W

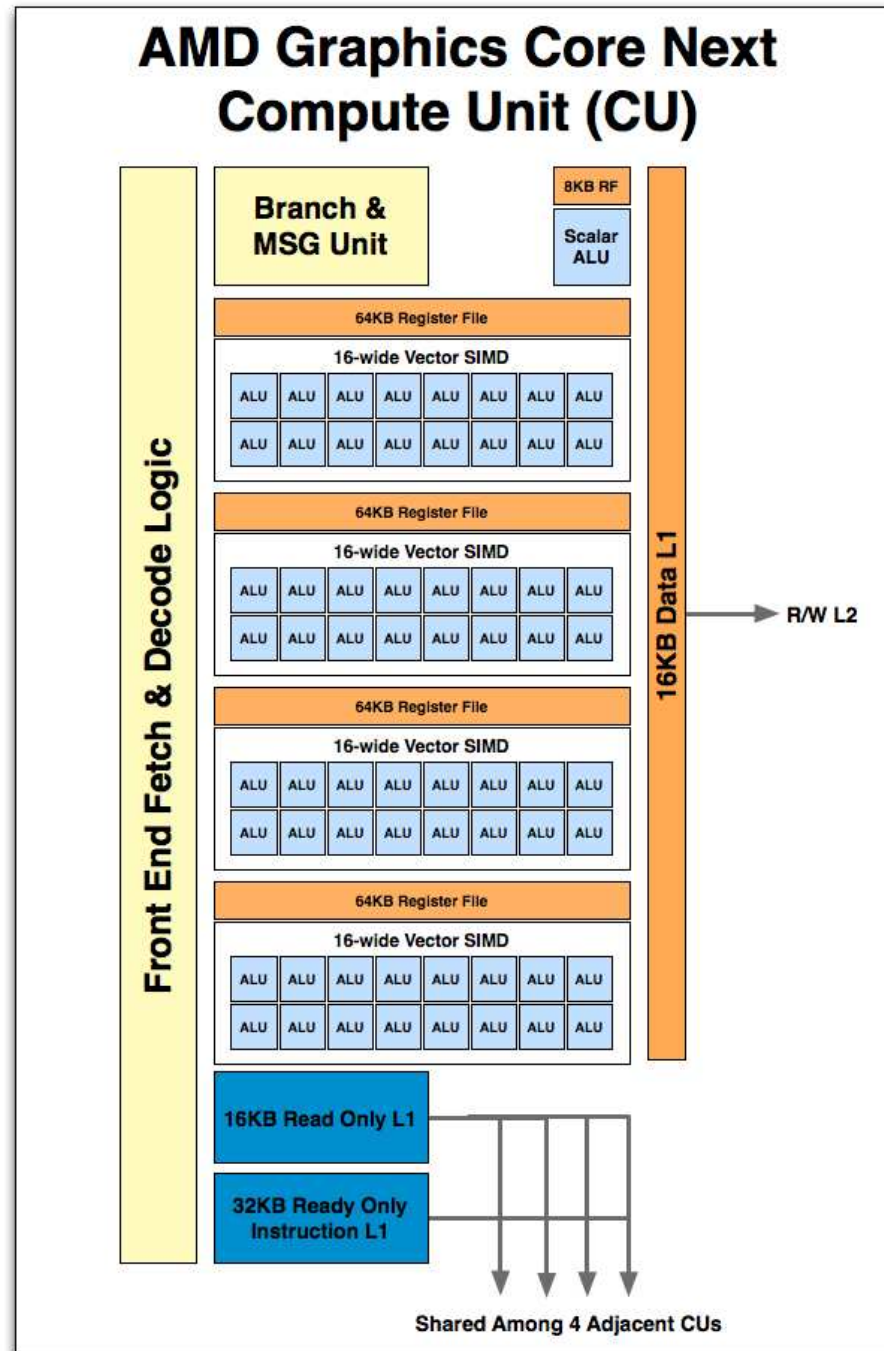
Arquitetura Graphics Core Next (GCN) – AMD

Arquitetura GCN – AMD

- AMD 7970 – GCN
- 32 unidades computacionais
- 64 elementos de processamento por unidade
 - do tipo escalar
- $\sim 4,3$ TF/s em precisão simples
- $\sim 1,1$ TF/s em precisão dupla (1/4)
- Memória: 288GB/s
- Consumo: ~ 250 W
- 4,3 Bilhões de transistores



Unidade Computacional



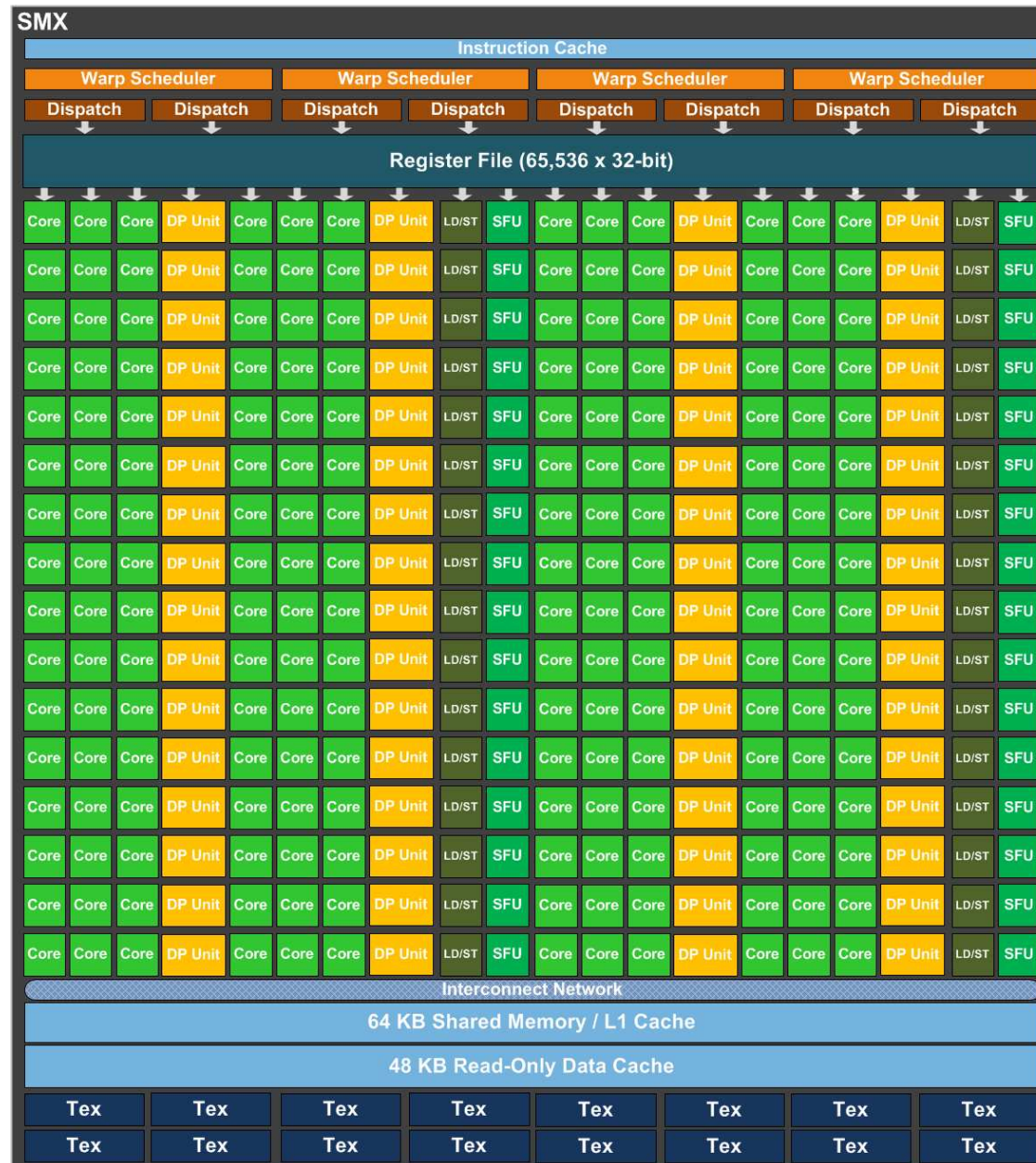
Arquitetura Kepler – Nvidia

Arquitetura Kepler

- Tesla K20X – *Kepler*
- 14 unidades computacionais
- 192 elementos de processamento por unidade
 - do tipo escalar
- $\sim 3,9$ TF/s em precisão simples
- $\sim 1,3$ TF/s em precisão dupla (1/3)
- Memória: 250GB/s
- Consumo: ~ 235 W
- 7,1 Bilhões de transistores



Unidade Computacional



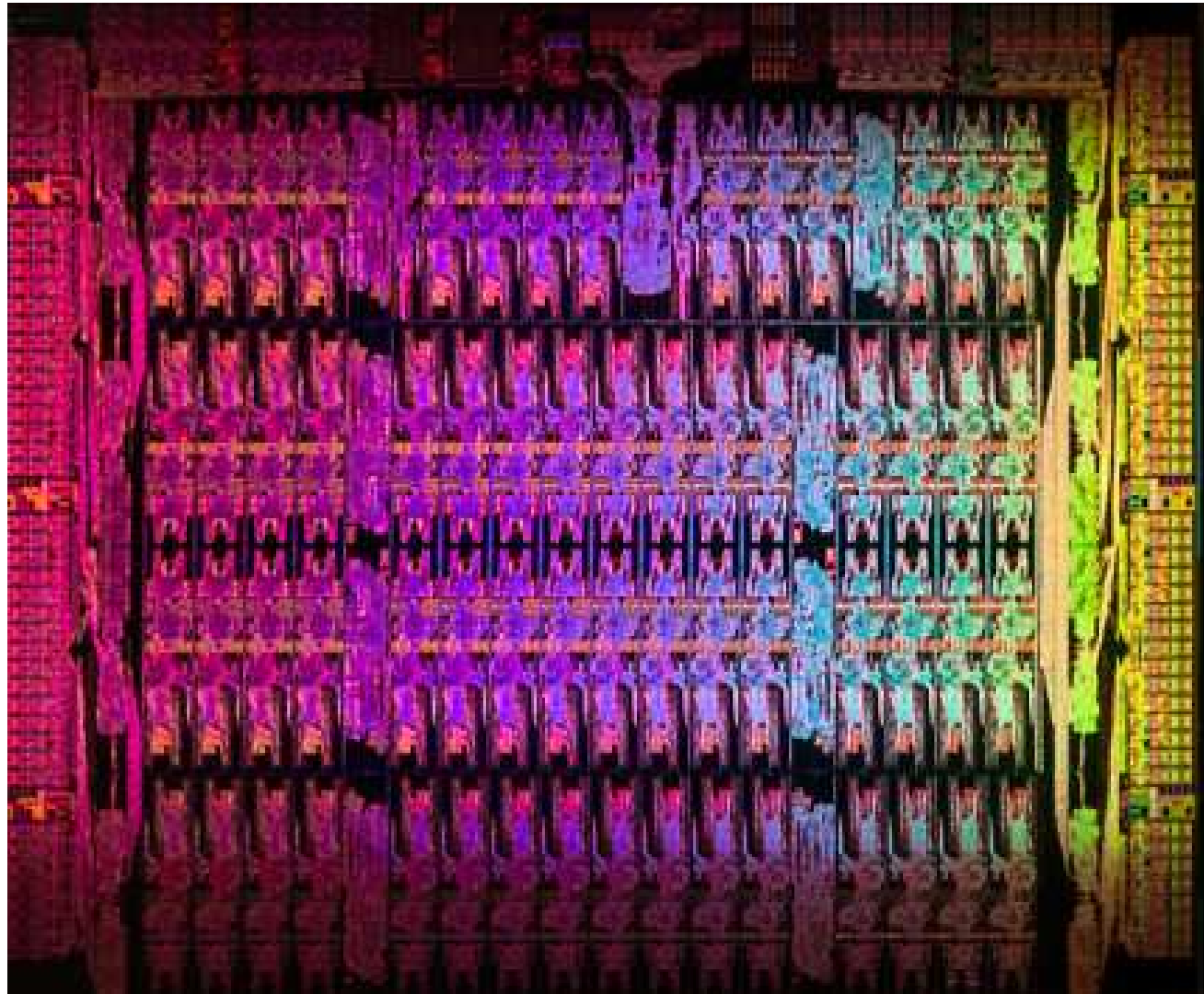
Arquitetura MIC – Intel

Arquitetura MIC

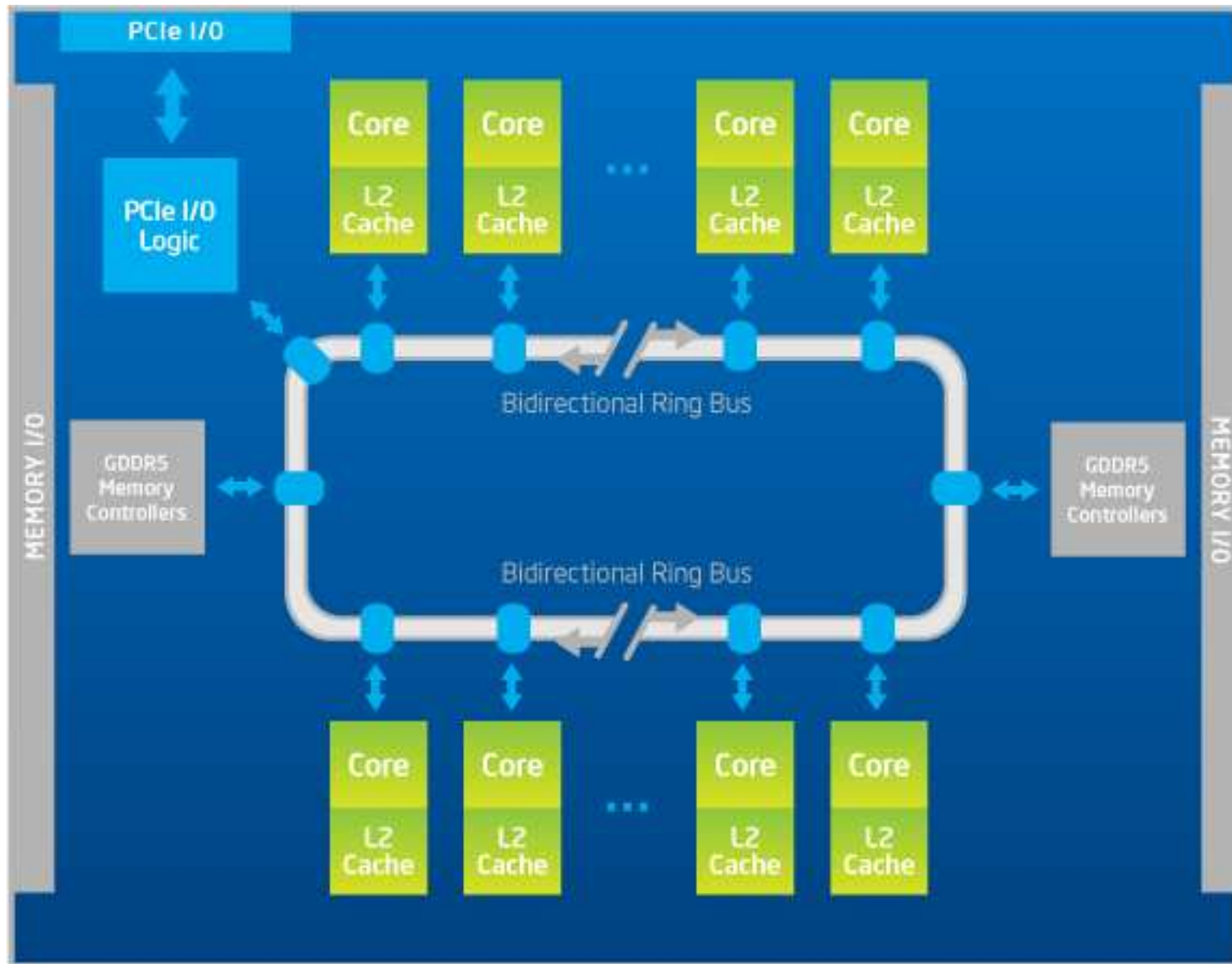


Arquitetura MIC

- Intel Phi – *MIC*
- 61 unidades computacionais
 - cada um do tipo SIMD (512-bit)
- $\sim 2,4$ TF/s em precisão simples
- $\sim 1,2$ TF/s em precisão dupla (1/2)
- Memória: 352GB/s
- Consumo: ~ 300 W

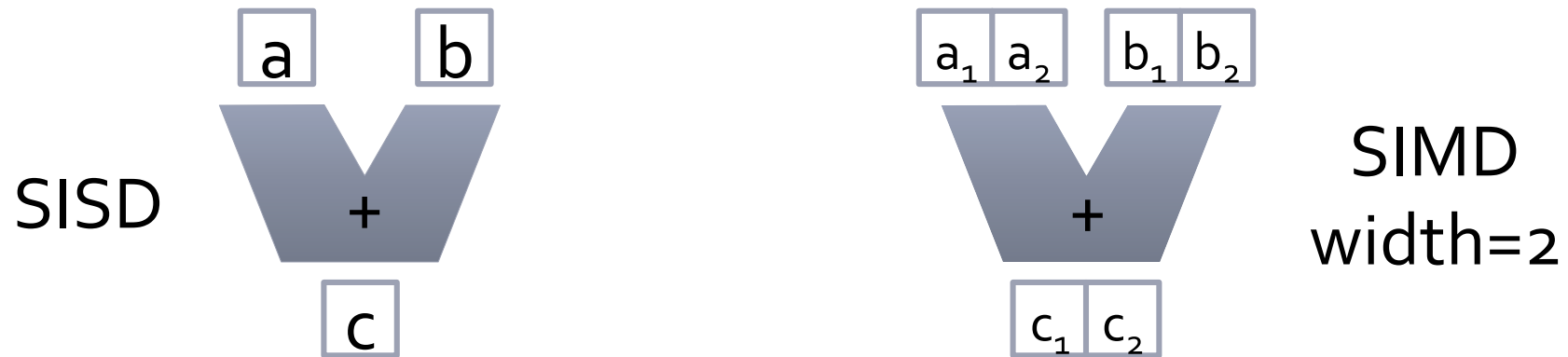


Arquitetura MIC



Modelo SIMD

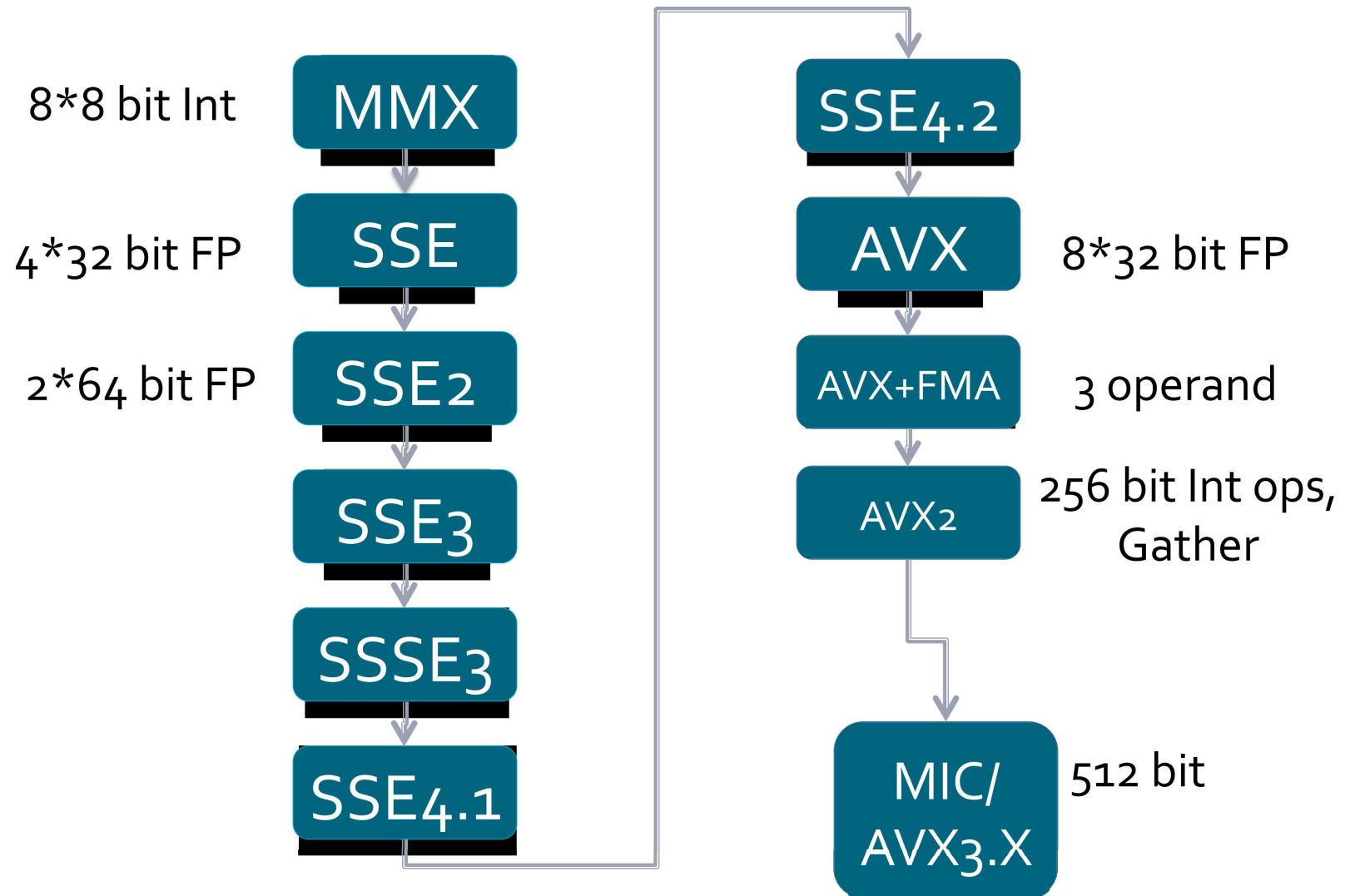
Modelo SIMD



SIMD — Single Instruction Multiple Data

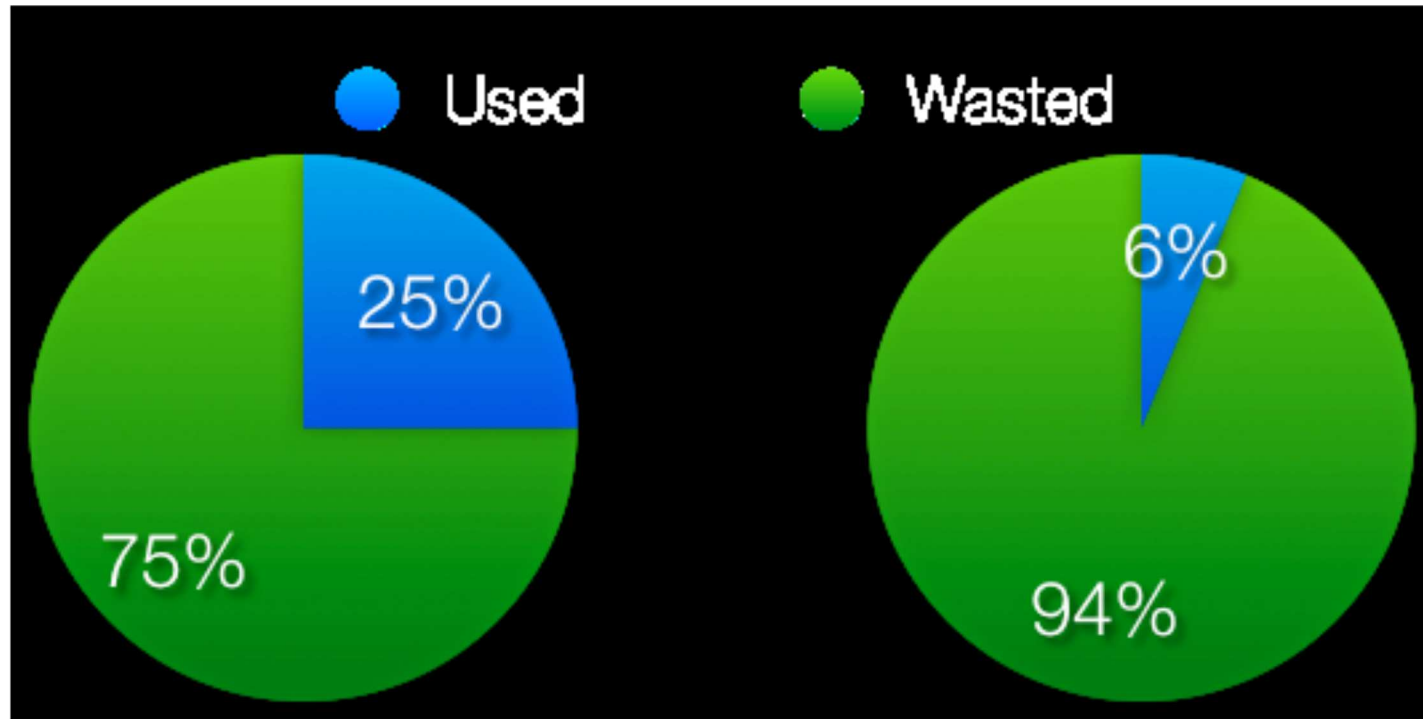
- As arquiteturas SIMD fazem uso do paralelismo de dados
- É interessante por questões de *área* e *energia*:
 - ◆ Elimina necessidade de unidades de controle
- Paralelismo é exposto para programador e compilador

Modelo SIMD



Modelo SIMD

Quanto se perde ao negligenciá-lo?



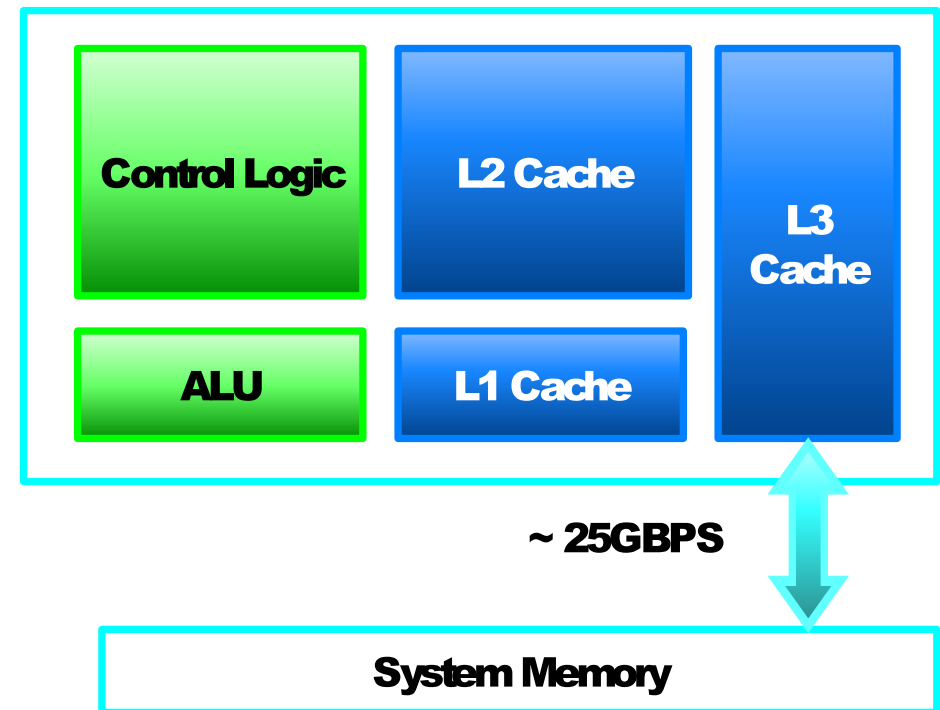
4 way SIMD (SSE)

16 way SIMD (Phi, AVX 3.x)

CPU versus GPU

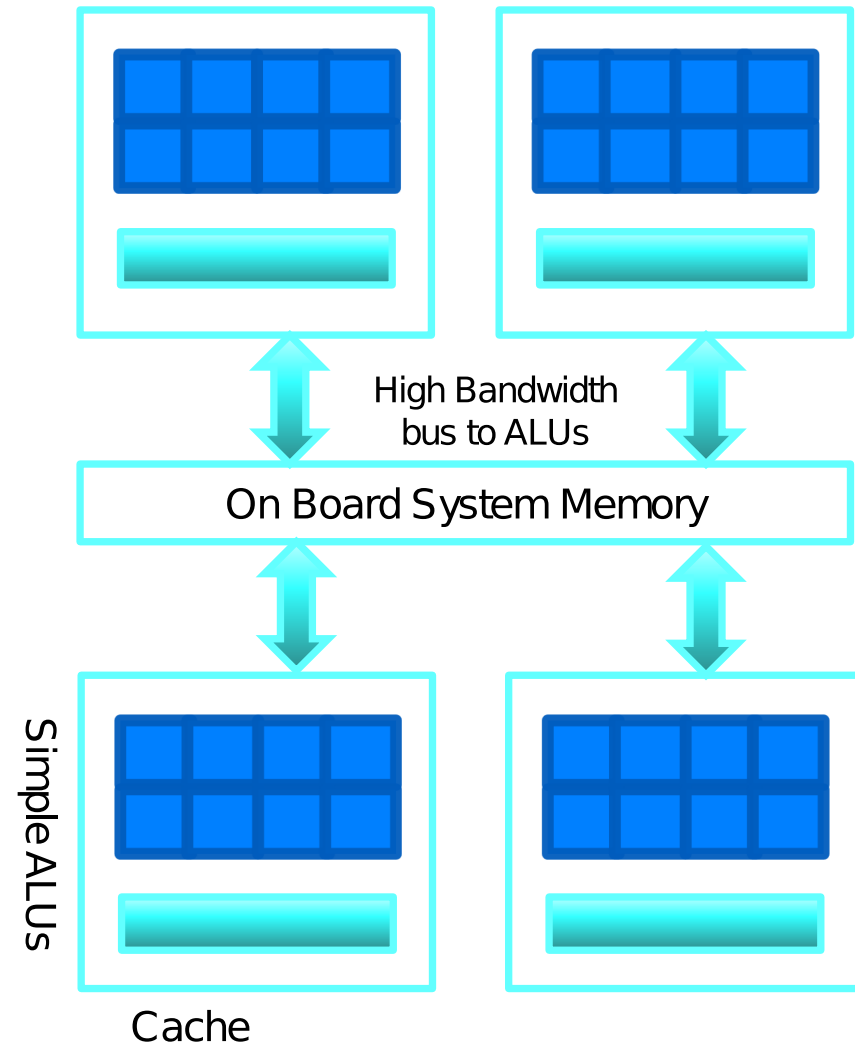
Arquitetura CPU Convencional

- Espaço dedicado à *unidade de controle* em vez de ALUs
- São otimizadas para minimizar a latência de um único *thread*
 - lida eficientemente com controle de fluxo
- Usa vários níveis de cache para encobrir latência
- Unidade de controle para reordenar execução, prover paralelismo de instruções e minimizar interrupções no pipeline



Arquitetura GPU Moderna

- Menos espaço dedicado à *unidade de controle e caches*
- Latência encoberta por alternância de *threads*
- Grande número de ALUs por unidade computacional
 - cada unidade contendo um pequeno cache
- Memória com grande largura de banda
 - ~ 300 GB/s para suprir várias ALUs simultaneamente



Diferenças cruciais: CPU x GPU

Arquitetura:

- **CPU:** MIMD (*Multiple Instruction Multiple Data*)
 - paralelismo de tarefas e dados
 - também possui paralelismo via instruções estendidas SIMD
 - mais flexível, propósito geral
- **GPU:** SIMD/SPMD (*Single Instruction/Program Multiple Data*)
 - paralelismo de dados
 - mais restrita (especializada), mas continuamente adquire características de propósito geral

Diferenças cruciais: CPU x GPU

Programação:

- **CPU:** facilmente programável
 - conceitualmente mais simples, foco essencialmente sequencial
 - maior disponibilidade e maturidade de linguagens e ferramentas de suporte (ex: depuração)
- **GPU:** programação menos direta
 - foco no paralelismo e escalabilidade
 - custo de engenharia de software (implementação, depuração, manutenção, etc.)
 - mais sensível ao projeto do algoritmo...
...ou, por outro lado, maior margem de otimização

Diferenças cruciais: CPU x GPU

Carga de trabalho:

- **CPU**: projetada para **reduzir a latência** na execução de uma tarefa:
 - baixa latência na execução de instruções e acesso à memória
 - uso intenso de memórias *cache* e outras tecnologias
- **GPU**: projetada para **aumentar a vazão** (*throughput*):
 - “cada pixel pode demorar quanto tempo for...
...desde que sejam processados vários ao mesmo tempo”

Diferenças cruciais: CPU x GPU

Processamento: o poder bruto de processamento da **GPU** é significativamente maior:

- Grande número de unidades computacionais: centenas ou milhares
- Aproveitamento dos recursos (transistores) em processadores mais simples:
 - implementação minimalista (ou inexistente): *unidades de controle, memórias cache, execução fora-de-ordem, predição de desvios, execução especulativa, etc.*

Mas o desempenho da **GPU** sofre sob *cargas de trabalhos irregulares*, por exemplo, com muitos desvios.

Diferenças cruciais: CPU x GPU

Memória:

■ CPU:

- em geral maior capacidade de armazenamento
- goza de acesso direto à memória do sistema

■ GPU:

- menor capacidade de armazenamento
- seu uso normalmente requer transferência prévia de dados da memória do sistema para o dispositivo

Diferenças cruciais: CPU x GPU

Escalabilidade:

■ CPU: menos escalável

- os processadores (núcleos) são complexos
- a arquitetura limita o número máximo viável de núcleos

■ GPU: mais escalável

- a expansão do número de processadores é “trivial”
- pode-se facilmente adicionar ao computador novos dispositivos ou atualizar os existentes

Perfil Ótimo de Carga em GPU

Recomendações

Válido para qualquer arquitetura de GPU moderna:

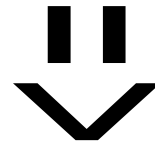
- Muitos (milhares) de *threads* independentes
 - uso de todas unidades computacionais
 - admite alternância de *threads* para encobrir latência
- Minimiza desvios de fluxo (baixa ramificação)
 - evita o problema da divergência
- Possui alta intensidade aritmética
 - razão *cômputo*/acesso à memória é alta
 - evita gargalo de acesso à memória

Fundamentos do OpenCL

Problema Ilustrativo: $\sqrt{\quad}$

Calcular a raiz quadrada de cada elemento de um vetor:

	0	1	2	3	4	5	...	n-1
float* X	0	1	2	3	4	5	...	n-1



	$\sqrt{0}$	$\sqrt{1}$	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{4}$	$\sqrt{5}$	$\sqrt{\dots}$	$\sqrt{n-1}$
float* Y	0	1	2	3	4	5	...	n-1

Problema Ilustrativo: $\sqrt{\quad}$

Solução sequencial:

```
void raiz( const float * x, float * y, int n )
{
    for( int i = 0; i < n; ++i )
    {
        y[i] = sqrt( x[i] );
    }
}
```

float* X

0	1	2	3	4	5	...	n-1
0	1	2	3	4	5	...	n-1



float* Y

$\sqrt{0}$	$\sqrt{1}$	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{4}$	$\sqrt{5}$	$\sqrt{\dots}$	$\sqrt{n-1}$
0	1	2	3	4	5	...	n-1

Problema Ilustrativo:

Solução paralela via OpenCL (*kernel*):

```
__kernel void raiz( __global const float * x, __global float * y )
{
    int i = get_global_id(0);
    y[i] = sqrt( x[i] );
}
```

float* X

0	1	2	3	4	5	...	n-1
0	1	2	3	4	5	...	n-1



float* Y

$\sqrt{0}$	$\sqrt{1}$	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{4}$	$\sqrt{5}$	$\sqrt{\dots}$	$\sqrt{n-1}$
0	1	2	3	4	5	...	n-1

Problema Ilustrativo: ✓

Solução sequencial:

```
void raiz( const float * x, float * y, int n )
{
    for( int i = 0; i < n; ++i )
    {
        y[i] = sqrt( x[i] );
    }
}
```

Solução paralela via OpenCL (*kernel*):

```
__kernel void raiz( __global const float * x, __global float * y )
{
    int i = get_global_id(0);
    y[i] = sqrt( x[i] );
}
```

Introdução

Código do Kernel e Hospedeiro

Existem duas hierarquias de códigos no OpenCL:

- O *kernel*:
 - tarefa executada paralelamente em um dispositivo computacional
 - implementado em C/C++ (baseado na especificação C11/C++14)

```
__kernel void f(...)  
{  
    ...  
}
```

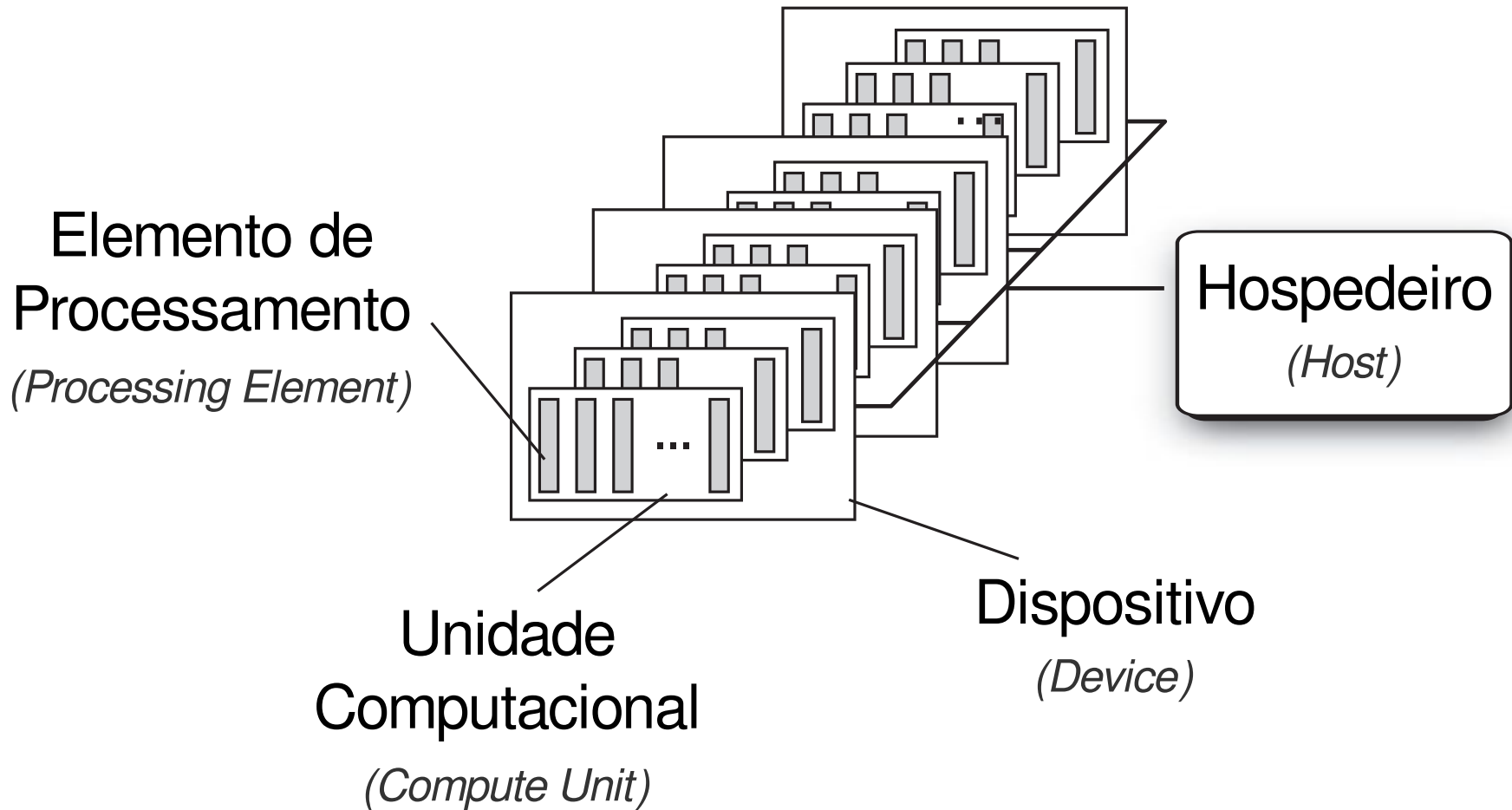
Arquitetura do OpenCL

Modelos

O OpenCL pode ser conceitualmente visto sob quatro ângulos:

- Modelo de *plataforma*
- Modelo de *execução*
- Modelo de *memória*
- Modelo de *programação*

Modelo de Plataforma



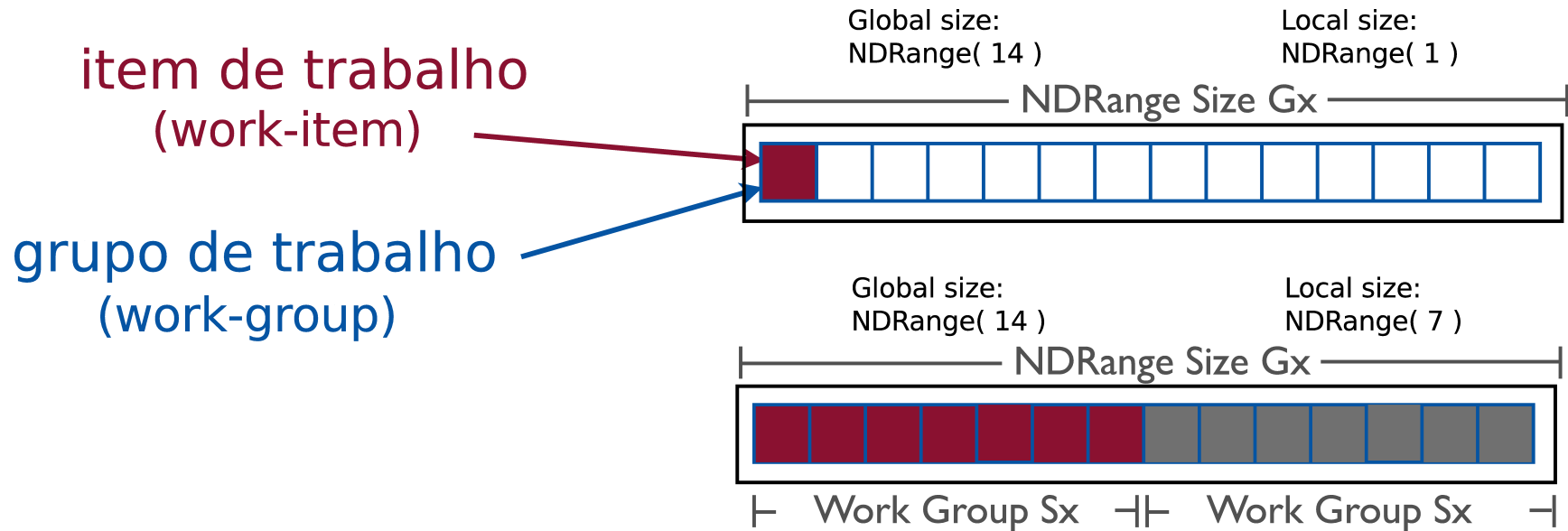
Modelo de Execução

Baseia-se nos elementos:

- **Item de trabalho** (*work-item*):
 - uma *instância* do *kernel* em execução
 - *unidade de execução* concorrente do OpenCL
 - possui identificadores *local* e *global* dentro de um *domínio de índices*

- **Grupo de trabalho** (*work-group*):
 - uma coleção de itens de trabalho
 - itens de trabalho de um mesmo grupo podem se *comunicar eficientemente e sincronizar*

Modelo de Execução



$$\text{NDRange Size} = \text{Global Size}$$

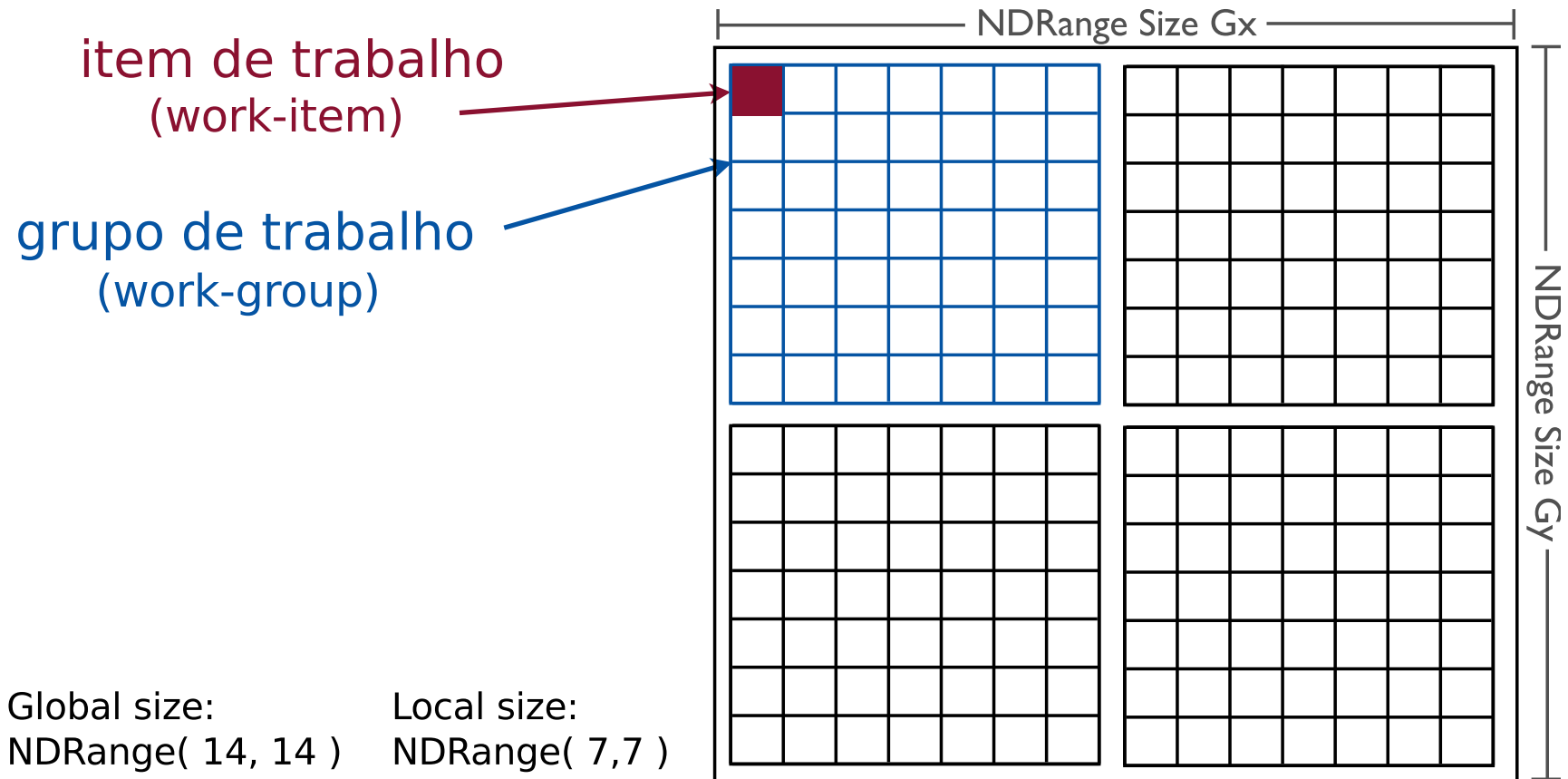
(Tamanho global)

$$\text{Work Group Size} = \text{Local Size}$$

(Tamanho do grupo de trabalho) (Tamanho local)

Domínio de índices unidimensional

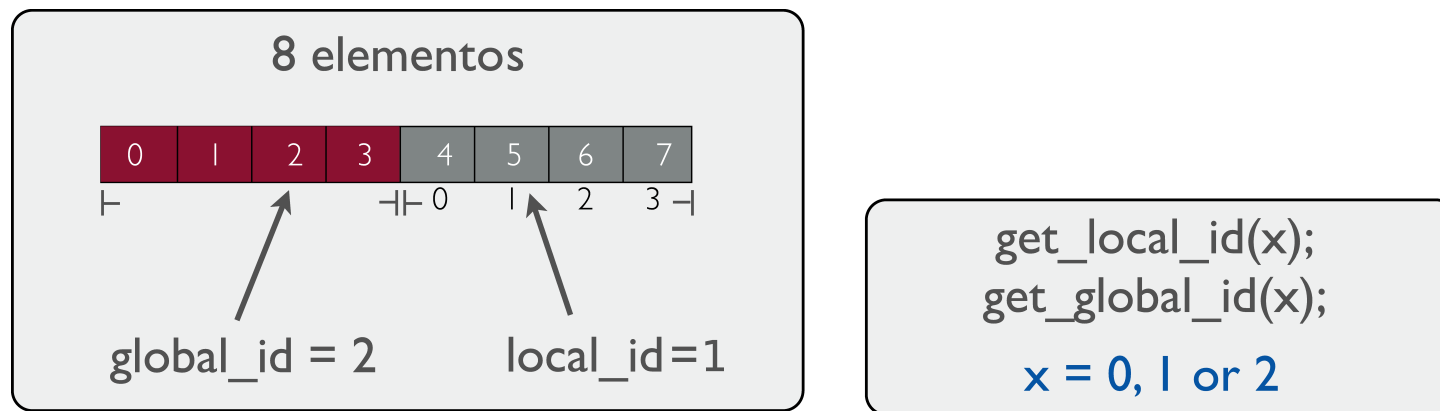
Modelo de Execução



Domínio de índices bidimensional

Modelo de Execução

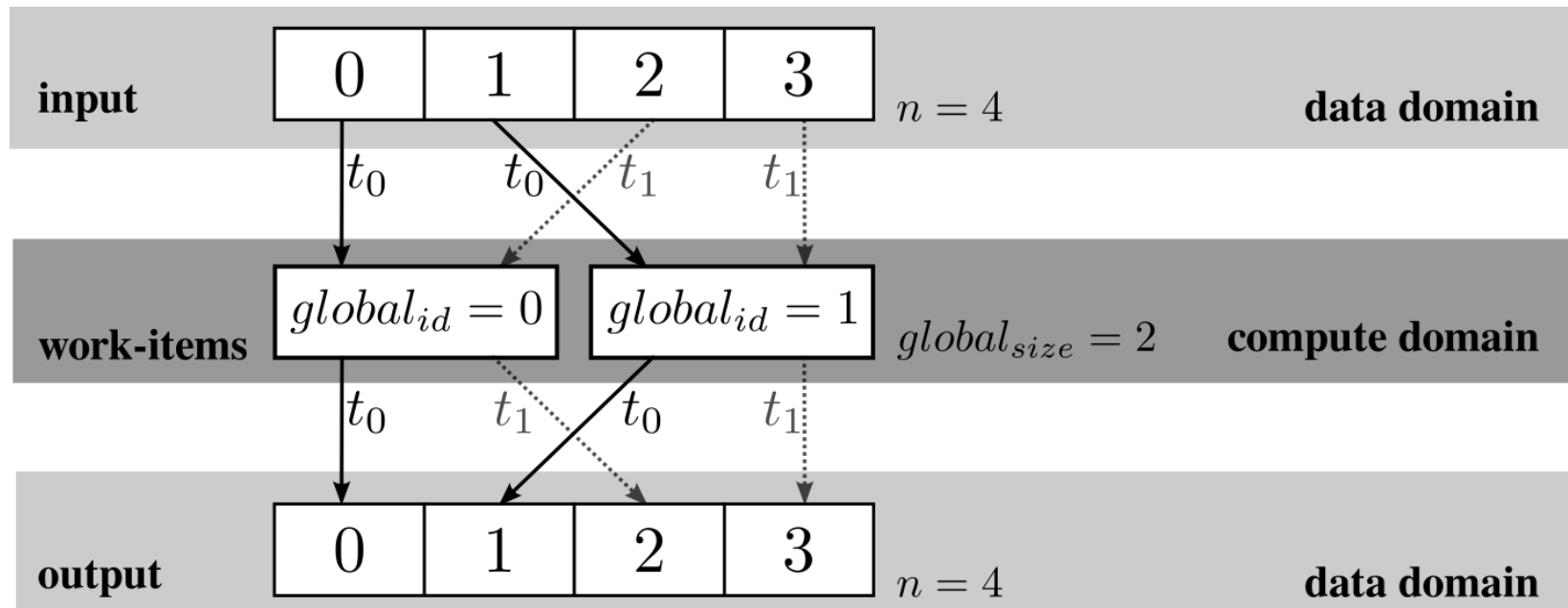
- Cada item de trabalho está “ciente” sobre qual elemento do problema ele está trabalhando
- Cada item de trabalho (e grupo) pode ser identificado dentro do *kernel*



Identificadores

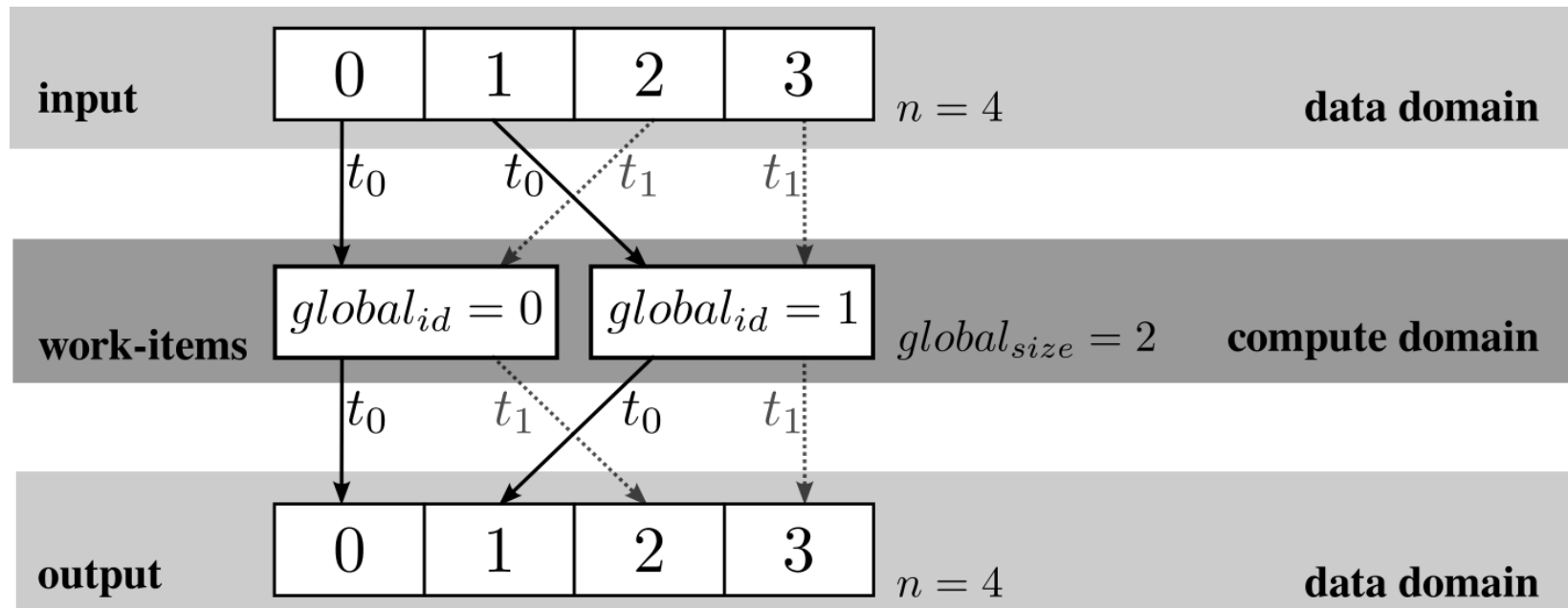
Modelo de Execução

- O domínio de índices é o mecanismo que conecta o *domínio de dados* ao *domínio de cálculo*



Modelo de Execução

- O domínio de índices é o mecanismo que conecta o *domínio de dados* ao *domínio de cálculo*

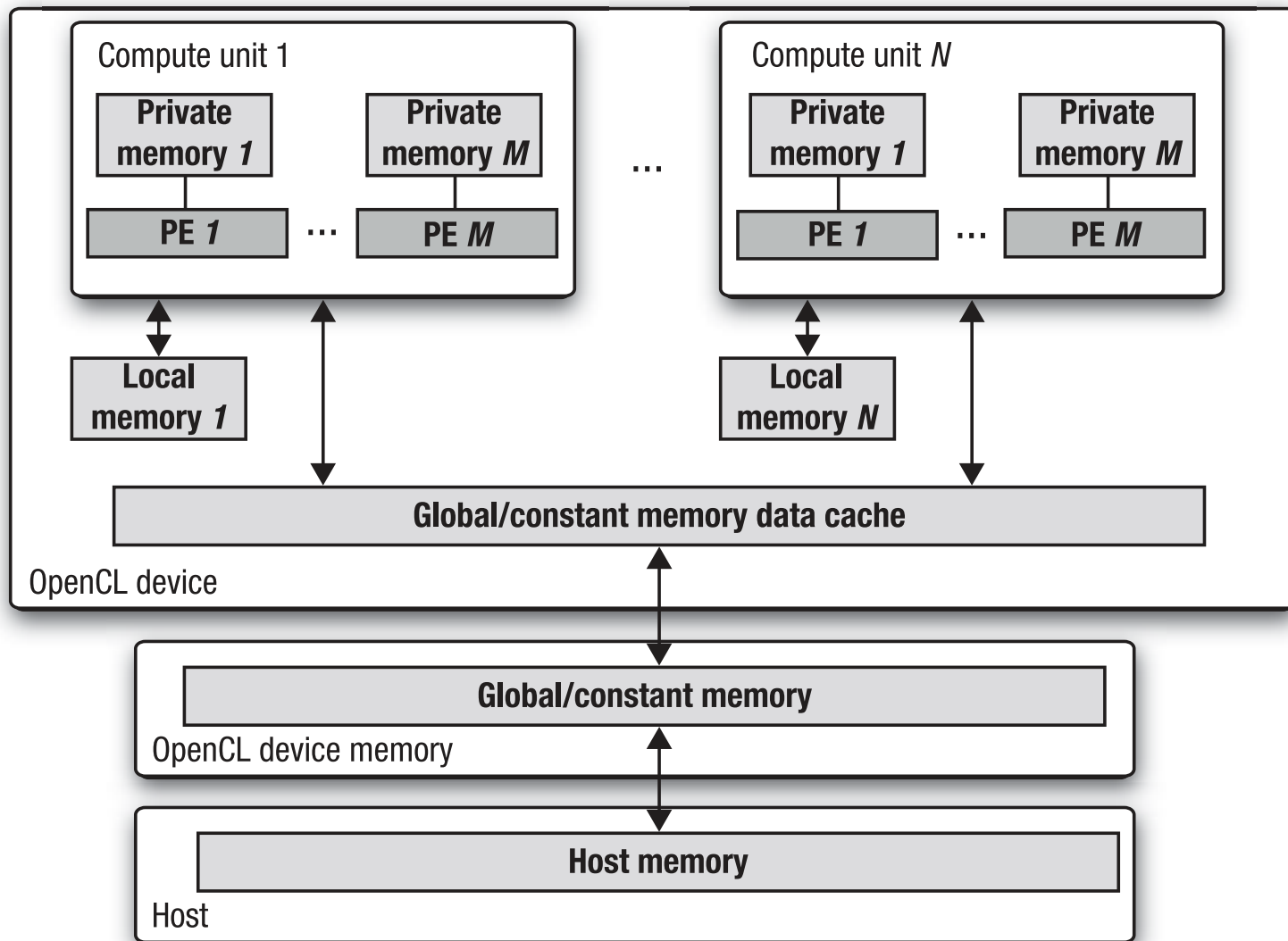


Pseudo-kernel:

```
for  $t \leftarrow 0$  to  $n/global\_size - 1$  do
   $i \leftarrow t \times global\_size + global\_id$ ;
   $output[i] \leftarrow \sqrt{input[i]}$ ;
end
```

Modelo de Memória

`--private`, `--local`, `--constant`, `--global`



Modelo de Programação

- Paralelismo de dados
 - modelo mais natural aos aceleradores (maior escalabilidade)
 - hierárquico: inter e intra grupo de trabalho
 - instruções vetoriais SIMD
- Paralelismo de tarefas

Dinâmica do OpenCL

Passos de Execução

Uma modelagem típica consiste em:

1. **Inicialização**
2. **Preparação da memória** (leitura e escrita)
3. **Execução**

Passos de Execução

1. Inicialização

- Descobrir e escolher plataformas e dispositivos
- Criar o contexto de execução
- Criar a fila de comandos para um dispositivo
- Carregar o programa, compilá-lo e gerar o *kernel*

Passos de Execução

1. Inicialização

- Descobrir e escolher plataformas e dispositivos
- Criar o contexto de execução
- Criar a fila de comandos para um dispositivo
- Carregar o programa, compilá-lo e gerar o *kernel*

2. Preparação da memória (leitura e escrita)

Passos de Execução

1. Inicialização

- Descobrir e escolher plataformas e dispositivos
- Criar o contexto de execução
- Criar a fila de comandos para um dispositivo
- Carregar o programa, compilá-lo e gerar o *kernel*

2. Preparação da memória (leitura e escrita)

3. Execução

- Transferência de dados para o dispositivo
- Execução do *kernel*: definição dos argumentos e trabalho/particionamento
- Espera pela finalização da execução do *kernel*
- Transferência dos resultados para o hospedeiro

Problema Ilustrativo: ✓

Problema Ilustrativo: $\sqrt{\quad}$

Calcular a raiz quadrada de cada elemento de um vetor

	0	1	2	3	4	5	...	n-1
float* X	0	1	2	3	4	5	...	n-1



	$\sqrt{0}$	$\sqrt{1}$	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{4}$	$\sqrt{5}$	$\sqrt{\dots}$	$\sqrt{n-1}$
float* Y	0	1	2	3	4	5	...	n-1

Cabeçalho

```
// Habilita disparar exceções C++
```

```
#define __CL_ENABLE_EXCEPTIONS
```

```
// Cabeçalho OpenCL para C++
```

```
#include <cl.hpp>
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <utility>
```

```
#include <cstdlib>
```

```
using namespace std;
```

Kernel

```
const char * kernel_str =  
    "__kernel void "  
    "raiz( __global const float * x, __global float * y ) "  
    "{ "  
    "    int i = get_global_id(0); "  
    "    y[i] = sqrt( x[i] ); "  
    "} ";
```

Entrada

```
int main( int argc, char* argv[] )
{
    // Dados de entrada: vetor de número reais
    const int elementos = atoi( argv[1] );

    float *X = new float[elementos];
    for( int i = 0; i < elementos; ++i ) X[i] = i;
    float *Y = new float[elementos];
```

Inicialização

```
//    Descobrir e escolher as plataformas e dispositivos
vector<cl::Platform> plataformas;
vector<cl::Device> dispositivos;

cl::Platform::get( &plataformas ); // plataformas
plataformas[0].getDevices( CL_DEVICE_TYPE_ALL, &dispositivos ); // dispositivos

//    Criar o contexto
cl::Context contexto( dispositivos );

//    Criar a fila de comandos para um dispositivo
cl::CommandQueue fila( contexto, dispositivos[0] );

//    Carregar o programa, compilá-lo e gerar o kernel
cl::Program::Sources fonte( 1, make_pair( kernel_str, strlen( kernel_str ) ) );
cl::Program programa( contexto, fonte );

programa.build();

cl::Kernel kernel( programa, "raiz" );
```

Preparação da Memória

```
cl::Buffer bufferX( contexto, CL_MEM_READ_ONLY, elementos * sizeof( float ) );  
cl::Buffer bufferY( contexto, CL_MEM_WRITE_ONLY, elementos * sizeof( float ) );
```

Execução

// Transferência de dados para o dispositivo

```
fila.enqueueWriteBuffer( bufferX, CL_TRUE, 0, elementos * sizeof( float ), X );
```

// Execução do kernel: definição dos argumentos e trabalho/particionamento

```
kernel.setArg( 0, bufferX );
```

```
kernel.setArg( 1, bufferY );
```

```
fila.enqueueNDRangeKernel( kernel, cl::NDRange(),  
                           cl::NDRange( elementos ), cl::NDRange() );
```

// Espera pela finalização da execução do kernel

```
fila.finish();
```

// Transferência dos resultados para o hospedeiro

```
fila.enqueueReadBuffer( bufferY, CL_TRUE, 0, elementos * sizeof( float ), Y );
```

Finalização

```
// Impressão do resultado
```

```
for( int i = 0; i < elementos; ++i ) cout << '[' << Y[i] << ' '; cout << endl;
```

```
// Limpeza
```

```
delete[] X, Y;
```

```
return 0;
```

```
}
```

```

#define __CL_ENABLE_EXCEPTIONS
#include <cl.hpp>
#include <iostream>
#include <vector>
#include <utility>
#include <cstdlib>
using namespace std;
const char * kernel_str =
    "__kernel void "
    "raiz( __global const float * x, __global float * y ) "
    "{ "
    "    int i = get_global_id(0); "
    "    y[i] = sqrt( x[i] ); "
    "} ";
int main( int argc, char* argv[] )
{
    const int elementos = atoi( argv[1] );
    float *X = new float[elementos];
    for( int i = 0; i < elementos; ++i ) X[i] = i;
    float *Y = new float[elementos];
    // --- Inicialização:
    vector<cl::Platform> plataformas;
    vector<cl::Device> dispositivos;
    cl::Platform::get( &plataformas );
    plataformas[0].getDevices( CL_DEVICE_TYPE_ALL, &dispositivos );
    cl::Context contexto( dispositivos );
    cl::CommandQueue fila( contexto, dispositivos[0] );
    cl::Program::Sources fonte( 1, make_pair( kernel_str, strlen( kernel_str ) ) );
    cl::Program programa( contexto, fonte );
    programa.build();
    cl::Kernel kernel( programa, "raiz" );
    // --- Preparação da memória:
    cl::Buffer bufferX( contexto, CL_MEM_READ_ONLY, elementos * sizeof( float ) );
    cl::Buffer bufferY( contexto, CL_MEM_WRITE_ONLY, elementos * sizeof( float ) );
    // --- Execução:
    fila.enqueueWriteBuffer( bufferX, CL_TRUE, 0, elementos * sizeof( float ), X );
    kernel.setArg( 0, bufferX );
    kernel.setArg( 1, bufferY );
    fila.enqueueNDRangeKernel( kernel, cl::NDRange(), cl::NDRange( elementos ), cl::NDRange() );
    fila.finish();
    fila.enqueueReadBuffer( bufferY, CL_TRUE, 0, elementos * sizeof( float ), Y );
    for( int i = 0; i < elementos; ++i ) cout << '[' << Y[i] << ']'<< endl;
    delete[] X, Y;
    return 0;
}

```

Compilação e Execução

OpenCL é uma especificação; implementações (plataformas) são fornecidas independentemente:

- AMD
 - Suporte às GPUs AMD + CPUs AMD e Intel
- Intel
 - Suporte às CPUs, GPUs e MICs Intel
- Nvidia
 - Suporte às GPUs Nvidia
- Outras: IBM, etc.

Compilação e Execução

No GNU/Linux:

■ Compilação:

```
g++ -o <out> <c++ source> -I<OpenCL-include-dir> -L<OpenCL-libdir> -lOpenCL
```

```
g++ -o ex ex.cc -I/usr/include/CL -lOpenCL
```

```
g++ -o ex ex.cc -I. -lOpenCL
```

■ Execução:

```
./ex <n>
```

```
./ex 10
```

```
[0] [1] [1.41421] [1.73205] [2] [2.23607] [2.44949] [2.64575] [2.82843] [3]
```

O Kernel OpenCL

Kernel OpenCL

- Escrito em uma linguagem de programação conhecida como *OpenCL C/C++*
 - derivada da especificação C11/C++14
 - modificações para comportar arquiteturas heterogêneas

Linguagem OpenCL C/C++

Exclusões:

- Recursividade
- Ponteiros para funções (e funções virtuais)
- Vetores (*arrays*) de tamanho variável
- Ponteiros para ponteiros como argumentos
- Exceções C++ (*throw, catch*)

Linguagem OpenCL C/C++

Extensões

- Qualificadores de espaço de memória

global, constant, local, private; ou
--global, --constant, --local, --private

- Biblioteca nativa de funções e constantes:

lógicas, aritméticas, relacionais, trigonométricas,
atômicas, etc.

- Tipos vetoriais

Notação: `tipo<n>`, com $n = 1, 2, 4, 8, 16$

Ex: `int4, float8, short2, uchar16`

Linguagem OpenCL C/C++

Extensões (cont.)

■ Operações vetoriais

- entre vetores com mesmo número de componentes
- entre vetores e escalares

```
float4 v = (float4)(1.0, 2.0, 3.0, 4.0);  
float4 u = (float4)(1.0);  
float4 v2 = v * 2;  
float4 t = v + u;
```

Linguagem OpenCL C/C++

Funções de identificação

- Item/grupo de trabalho:

`get_global_id(dim)`

`get_local_id(dim)`

`get_group_id(dim)`

- Domínio de índices:

`get_work_dim()`

`get_global_size(dim)`

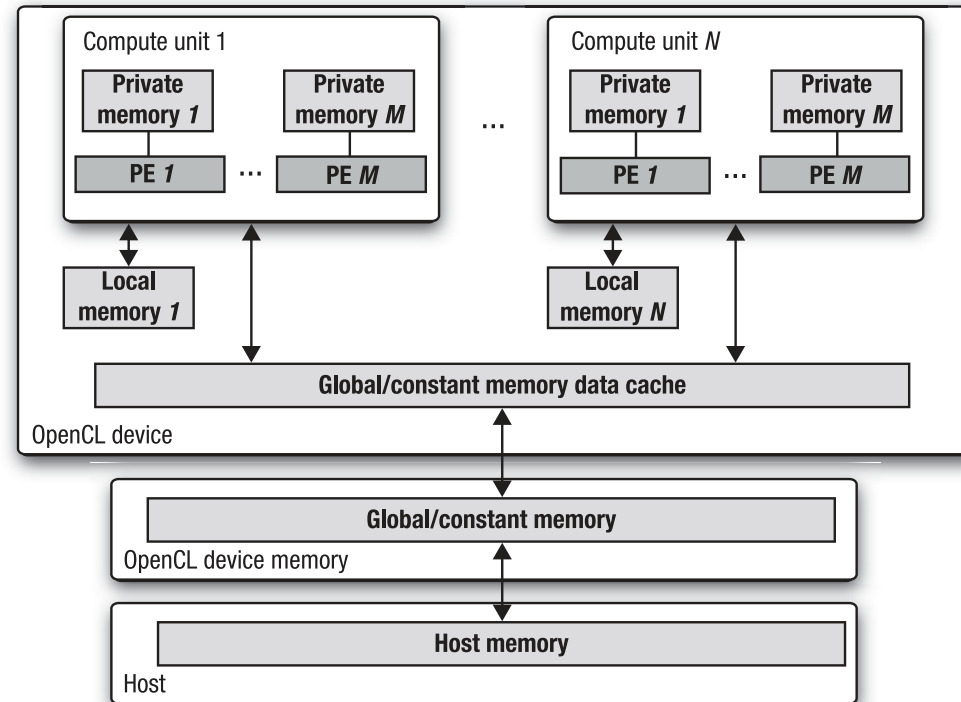
`get_local_size(dim)`

`get_num_groups(dim)`

`get_global_offset(dim)`

Modelo de Memória

Modelo de Memória



- *global*: acessível por todos itens de trabalho
- *constant*: acesso global, mas somente leitura
- *local*: somente acessível pelos itens dentro de um mesmo grupo de trabalho
- *private*: somente acessível pelo item de trabalho

Escopo de Alocação/Acesso à Memória

Memória	Hospedeiro		Kernel	
	<i>Alocação</i>	<i>Acesso</i>	<i>Alocação</i>	<i>Acesso</i>
<i>global</i>	dinâmica	leitura/escrita	–	leitura/escrita
<i>constant</i>	dinâmica	leitura/escrita	estática	leitura
<i>local</i>	dinâmica	–	estática	leitura/escrita
<i>private</i>	–	–	estática	leitura/escrita

Declarações de Variáveis no Kernel

- *global*: não permitida
 - visibilidade global; pode ser ineficiente gerenciar o endereçamento
- *constant*: a definição deve acompanhar a declaração
- *local*: não pode ser definida na declaração
- *private*: é o escopo padrão

```
kernel void f()  
{  
    __constant float c = 3.1415;    // constante  
    __local int loc[16];            // local  
    int i;                          // privada  
    ...
```

Declarações dos Argumentos do Kernel

Declaração:

```
kernel void f( __global const float * glc,  
               __global int * gl,  
               __constant float * cnt,  
               __local uint * loc,  
               float s )  
  
{ ... }
```

Sintaxe de definição:

```
setArg( índice, objeto );  
setArg( índice, tamanho, ponteiro );
```

Definição:

```
setArg( 0, bufferX );  
setArg( 1, bufferY );  
setArg( 2, bufferZ );  
setArg( 3, sizeof( uint ) * num_elementos, NULL );  
setArg( 4, (float) 3.1415 );
```

Consistência de Memória e Sincronia

Introdução

Consistência de memória diz respeito à correta visibilidade, em tempo de execução, de conteúdo de memória entre os itens de trabalho:

- não basta conhecer onde o conteúdo será armazenado; é preciso garantir que um item de trabalho leia corretamente os valores escritos pelos demais

O OpenCL adota um modelo *relaxado* de consistência de memória:

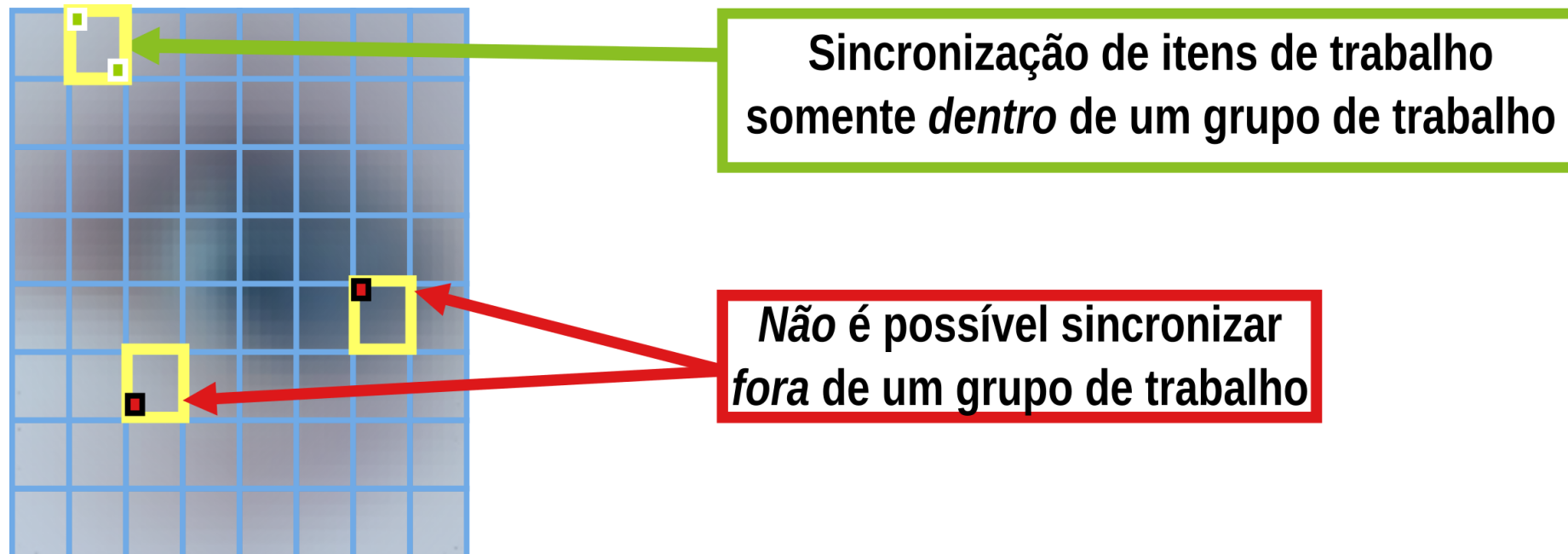
- dependendo do espaço de memória, a consistência só é obtida em pontos de sincronização

Modelo de Execução do OpenCL

- *Itens de trabalho* são executados nos *elementos de processamento*
- Cada *grupo de trabalho* é executado em uma *única unidade computacional*
 - diferentes grupos de trabalho são executados *independentemente*
 - na CPU uma unidade computacional é mapeada em um *núcleo* (explora o vetor SIMD)
 - na GPU ela é mapeada em uma coleção de *elementos de processamento*

Modelo de Execução do OpenCL

- “Não há” sincronia *global*
- Apenas itens de trabalho de um mesmo grupo podem sincronizar entre si



Modelo de Execução do OpenCL

Razões em favor da inexistência de sincronia global:

- Tácita, induzir um melhor particionamento do problema:
 - sincronia global implica em menor escalabilidade
- Suporte a dispositivos heterogêneos:
 - com sincronia global uma determinada arquitetura deveria ser capaz de gerenciar/executar *todos* os grupos de trabalho *concorrentemente*

Escalabilidade!

Consistência por Escopo de Memória

- Memória **privada** (*private*):
consistência *garantida*
- Memória **constante** (*constant*):
consistência *garantida*
(não há modificação de conteúdo)
- Memória **local** e **global**:
consistência *relaxada* entre itens de trabalho
requer sincronismo explícito

Primitiva de Sincronia

Primitiva de Sincronia

Itens de trabalho de um mesmo grupo são sincronizados—e a consistência garantida—usando-se no *kernel* a primitiva:

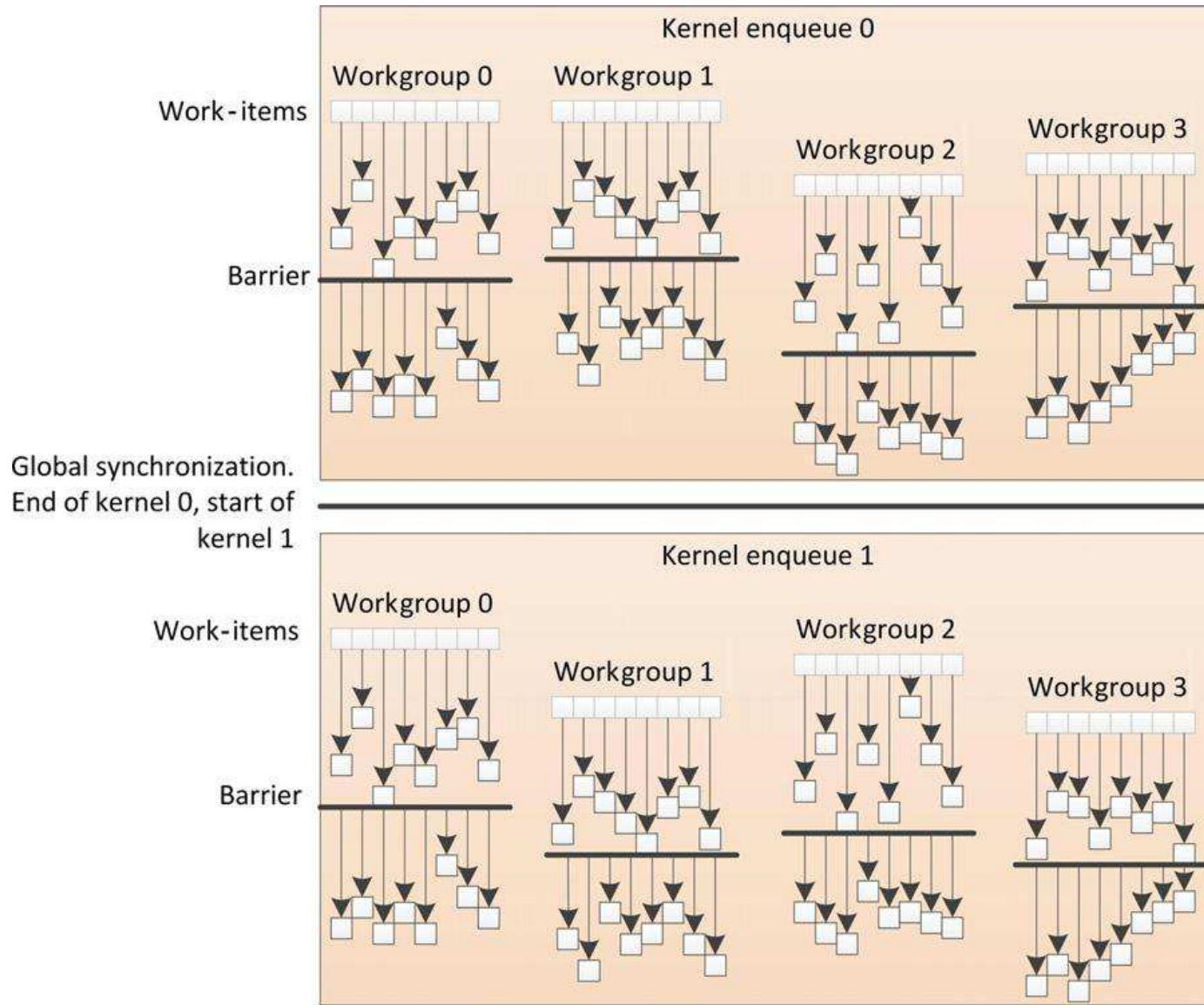
```
void barrier( <escopo> )
```

Onde *escopo* pode ser:

- CLK_LOCAL_MEM_FENCE: escopo *local*
- CLK_GLOBAL_MEM_FENCE: escopo *global*
- ou CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE

Todos os itens de trabalho de *um grupo* devem atingir este ponto do *kernel* para que a execução continue.

Primitiva de Sincronia



Exemplo Ilustrativo

Exemplo Ilustrativo

```
kernel void f()  
{  
    int i = get_global_id(0);  
    __local int x[10];  
    x[i] = i;  
  
    if( i > 0 )  
        int y = x[i-1];  
}
```

Exemplo de acesso inconsistente

Exemplo Ilustrativo

```
kernel void f()  
{  
    int i = get_global_id(0);  
    __local int x[10];  
    x[i] = i;  
  
    barrier( CLK_LOCAL_MEM_FENCE );  
  
    if( i > 0 )  
        int y = x[i-1];  
}
```

Acesso consistente após ponto de sincronia

Pontos de Sincronia

Considerações:

- Sincronias afetam negativamente o desempenho:
 - itens de trabalho no ponto de sincronia aguardam ociosamente pelos demais
- Devem ser escolhidas com cautela:
 - se um item de trabalho (de um grupo) não atinge o *barrier* a execução para indefinidamente:

```
kernel void deadlock( global float * x ) {  
    int i = get_global_id(0);  
    if( i == 0 )  
        barrier( CLK_LOCAL_MEM_FENCE );  
    else  
        x[i] = i;  
}
```

Tomada de Tempo

Tomada de Tempo

A especificação OpenCL fornece mecanismos precisos para se medir o tempo de *execução* (e outras medidas) de comandos.

- Baseia-se em *eventos*
- É habilitado fornecendo-se o argumento
CL_QUEUE_PROFILING_ENABLE
quando cria-se a *fila de comandos*

Tomada de Tempo

```
cl::CommandQueue fila( contexto, dispositivo, CL_QUEUE_PROFILING_ENABLE );

cl::Event e_tempo;

fila.enqueueNDRangeKernel( kernel, cl::NDRange(), cl::NDRange( elementos ),
                           cl::NDRange() , NULL, &e_tempo );

fila.finish();

cl_ulong inicio, fim;
e_tempo.getProfilingInfo( CL_PROFILING_COMMAND_START, &inicio );
e_tempo.getProfilingInfo( CL_PROFILING_COMMAND_END, &fim );

double tempo_execucao_s = (fim - inicio)/1.0E9;
```

Consulta de Propriedades

Consulta de Propriedades

Plataformas:

- Nome da plataforma:

```
plataforma.getInfo<CL_PLATFORM_NAME>();
```

Dispositivos:

- Tipo do dispositivo:

```
dispositivo.getInfo<CL_DEVICE_TYPE>();
```

- Nome do dispositivo:

```
dispositivo.getInfo<CL_DEVICE_NAME>();
```

- Número de unidades computacionais:

```
dispositivo.getInfo<CL_DEVICE_MAX_COMPUTE_UNITS>();
```

Consulta de Propriedades

Memória dos dispositivos:

- Memória *global* alocável (`__global`):

```
dispositivo.getInfo<CL_DEVICE_MAX_MEM_ALLOC_SIZE>();
```

- Memória *local* alocável (`__local`):

```
dispositivo.getInfo<CL_DEVICE_LOCAL_MEM_SIZE>();
```

- Memória *constante* alocável (`__constant`):

```
dispositivo.getInfo<CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE>();
```

Dimensões máximas:

- Tamanho máximo local:

```
dispositivo.getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();
```

- Tamanho máximo em cada dimensão:

```
dispositivo.getInfo<CL_DEVICE_MAX_WORK_ITEM_SIZES>()[dim];
```

Computação Heterogênea: Modelagem de um Problema

Problema

Computar:

$$\mathbf{x} = 3\mathbf{x} - \sqrt{\mathbf{x}}$$

■ Tempo 1:

CPU computa paralelamente $\mathbf{y} = 3\mathbf{x}$

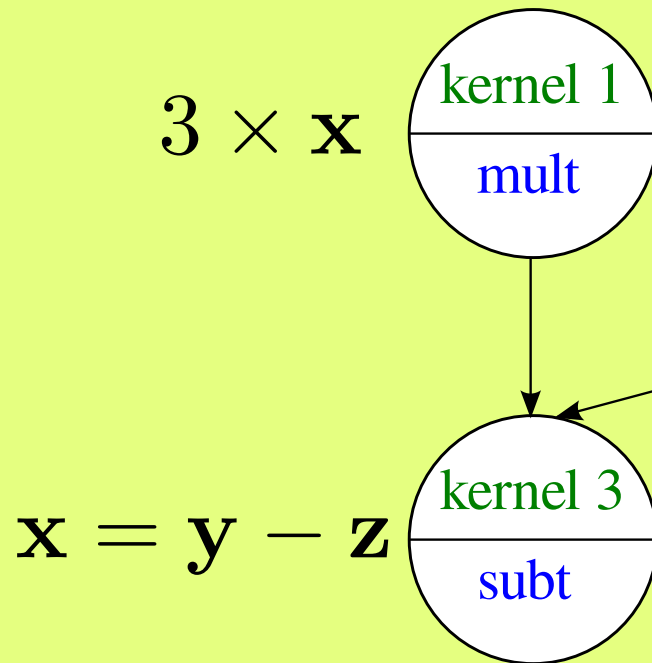
GPU computa paralelamente $\mathbf{z} = \sqrt{\mathbf{x}}$

■ Tempo 2:

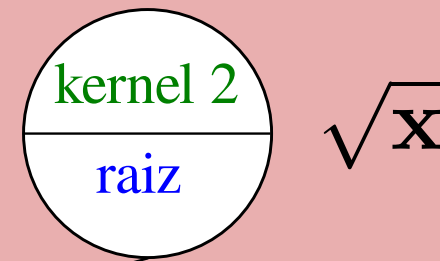
CPU computa paralelamente $\mathbf{x} = \mathbf{y} - \mathbf{z}$

Kernels e Dependências

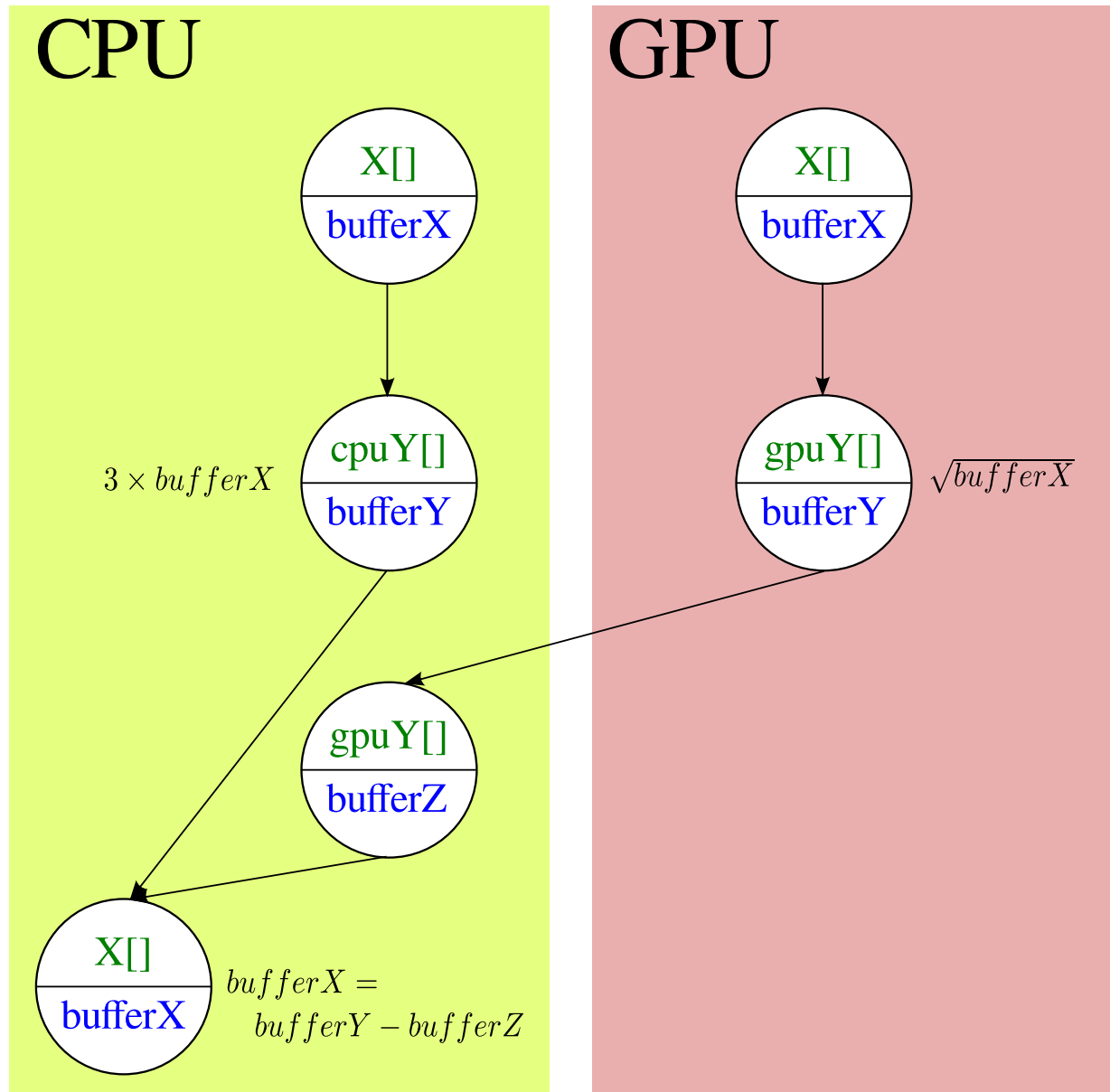
CPU



GPU



Buffers e Dependências



Kernels

```
__kernel void
raiz( __global const float * x, __global float * y )
{
    int i = get_global_id(0);
    y[i] = sqrt( x[i] );
}
```

```
__kernel void
mult( __global const float * x, __global float * y, float s )
{
    int i = get_global_id(0);
    y[i] = s * x[i];
}
```

```
__kernel void
subt( __global float * x, __global const float * y, __global const float * z )
{
    int i = get_global_id(0);
    x[i] = y[i] - z[i];
}
```

Memórias no Hospedeiro

```
// Aloca as memórias para os vetores X, cpuY e gpuY, e faz cada elemento do
// vetor X ter o valor do seu próprio índice
float*X = new float[elementos];
float*cpuY = new float[elementos];
float*gpuY = new float[elementos];

for(int i =0; i < elementos; ++i ) X[i] = i;
```

Plataformas, Contextos e Filas

```
// Descobrir e escolher as plataformas e dispositivos
vector<cl::Platform> plataformas;
vector<cl::Device> cpu_dispositivos, gpu_dispositivos;

// Descubra as plataformas instaladas no hospedeiro
cl::Platform::get( &plataformas );
// Descubra os dispositivos CPU e GPU: para simplificar, vamos procurar
// apenas os dispositivos da primeira plataforma (plataformas[0])
plataformas[0].getDevices( CL_DEVICE_TYPE_CPU, &cpu_dispositivos );
plataformas[0].getDevices( CL_DEVICE_TYPE_GPU, &gpu_dispositivos );

// Criar os contextos
cl::Context cpu_contexto( cpu_dispositivos );
cl::Context gpu_contexto( gpu_dispositivos );

// Criar as filas de comandos para cada arquitetura (o primeiro dispositivo)
cl::CommandQueue cpu_fila( cpu_contexto, cpu_dispositivos[0] );
cl::CommandQueue gpu_fila( gpu_contexto, gpu_dispositivos[0] );
```

Programas e Kernels

```
// Carregar os programas, compilá-los e gerar os kernels
cl::Program::Sources fonte(1,make_pair( kernel_str,strlen( kernel_str ) ) );
cl::Program cpu_programa( cpu_contexto, fonte );
cl::Program gpu_programa( gpu_contexto, fonte );

// Compila para todos os dispositivos associados a '[cpu|gpu]_programa'
// através de '[cpu|gpu]_contexto'
cpu_programa.build();
gpu_programa.build();

// Cria os objetos que representarão cada um dos três kernels
cl::Kernel kernel_mult( cpu_programa,"mult");
cl::Kernel kernel_subt( cpu_programa,"subt");
cl::Kernel kernel_raiz( gpu_programa,"raiz");
```

Buffers Iniciais

```
// Preparação da memória dos dispositivos (leitura e escrita}  
cl::Buffer cpu_bufferX( cpu_contexto, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,  
                        elementos *sizeof(float), X );  
cl::Buffer gpu_bufferX( gpu_contexto, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
                        elementos *sizeof(float), X );  
  
cl::Buffer cpu_bufferY( cpu_contexto, CL_MEM_READ_WRITE, elementos *sizeof(float) );  
cl::Buffer gpu_bufferY( gpu_contexto, CL_MEM_WRITE_ONLY, elementos *sizeof(float) );
```

Execução dos Kernels mult e raiz

```
// Execução dos kernels: definição dos argumentos e trabalho/particionamento
```

```
kernel_mult.setArg(0, cpu_bufferX );
```

```
kernel_mult.setArg(1, cpu_bufferY );
```

```
kernel_mult.setArg(2, float(3) );
```

```
kernel_raiz.setArg(0, gpu_bufferX );
```

```
kernel_raiz.setArg(1, gpu_bufferY );
```

```
// Paralelismo implícito: tamanho local é definido como "nulo"; a
```

```
// implementação é que vai decidir se divide em grupos e como dividi-los
```

```
cpu_fila.enqueueNDRangeKernel( kernel_mult, cl::NDRange(),
```

```
                                cl::NDRange( elementos ), cl::NDRange() );
```

```
gpu_fila.enqueueNDRangeKernel( kernel_raiz, cl::NDRange(),
```

```
                                cl::NDRange( elementos ), cl::NDRange() );
```

```
cpu_fila.flush();// força a execução dos comandos da fila
```

```
gpu_fila.flush();// força a execução dos comandos da fila
```

Coleta dos Resultados da GPU

```
// Transferência dos resultados da GPU para o hospedeiro (joga em gpuY)
// (comando bloqueante: CL_TRUE)
gpu_fila.enqueueReadBuffer( gpu_bufferY, CL_TRUE, 0,
                           elementos *sizeof(float), gpuY );

// Criar um buffer na CPU com os resultados oriundos da GPU
cl::Buffer cpu_bufferZ( cpu_contexto, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                       elementos *sizeof(float), gpuY );
```

Coleta dos Resultados Finais

```
// Execução do kernel final: definição dos argumentos e trabalho/particionamento
kernel_subt.setArg(0, cpu_bufferX );
kernel_subt.setArg(1, cpu_bufferY );
kernel_subt.setArg(2, cpu_bufferZ );

cpu_fila.enqueueNDRangeKernel( kernel_subt, cl::NDRange(),
                                cl::NDRange( elementos ), cl::NDRange() );

// Transferência dos resultados da CPU para o hospedeiro (joga em X)
// (comando bloqueante: CL_TRUE)
cpu_fila.enqueueReadBuffer( cpu_bufferX, CL_TRUE, 0, elementos *sizeof(float), X );
```

Impressão e Limpeza

```
// Impressão do resultado
```

```
for( int i =0; i < elementos; ++i ) cout <<'['<< X[i] <<']'; cout << endl;
```

```
// Limpeza (as variáveis específicas do OpenCL já são automaticamente destruídas)
```

```
delete[] X, cpuY, gpuY;
```

Referências

Referências

- **Heterogeneous Computing with OpenCL**

B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, D. Schaa

- **OpenCL Programming Guide**

A. Munshi, B. Gaster, T. G. Mattson, J. Fung, D. Ginsburg

- **OpenCL in Action**

Matthew Scarpino

- **The OpenCL Programming Book**

R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, J. Son, S. Miki

- **OpenCL Specification**

<http://www.khronos.org/opencl/>

“Basically it lets you use graphics processors to do computation,” he said. “It’s way beyond what Nvidia or anyone else has, and it’s really simple.”

Steve Jobs on OpenCL, NY Times interview, June 10 2008

Obrigado!